(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 416 568 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **90117042.3**

(51) Int. Cl.[5]: **G06F 15/60**, G06F 15/72

(22) Date of filing: **05.09.90**

(30) Priority: **07.09.89 US 404262**

(43) Date of publication of application:
13.03.91 Bulletin 91/11

(84) Designated Contracting States:
CH DE ES FR GB IT LI NL

(71) Applicant: **AMP-AKZO CORPORATION**
322 South Service Road
Melville, NY 11747(US)

(72) Inventor: **Williams, John D.**
237 Forrestal Drive
Bear, Delaware 19701(US)

(74) Representative: **Königseder-Egerer, Claudia D.**
Zugspitzstrasse 65
W-8104 Grainau(DE)

(54) Method of designing three dimensional electrical circuits.

(57) A computer aided design package is used to create a mathematical representation of a three-dimensional object. This object is defined as a set of surfaces oriented in space. A map of the flattened object is created by concatenating selected ones of the surfaces on a single plane. The outline of this map is then used in a computer aided circuit layout package as a printed circuit board on which an electrical circuit is placed and routed. The circuit is translated into a three-dimensional form corresponding to the surface of the object by translating and rotating the representation of the object to align each selected surface with the circuit description generated by the circuit layout package. The portion of the circuit corresponding to the surface is then transferred to a three-dimensional data structure having a format that is compatible with numerically controlled machining apparatus. This data structure is used to drive a numerically controlled phototool which creates a three-dimensional mask that may be used to print the circuit on the surface of the three-dimensional object.

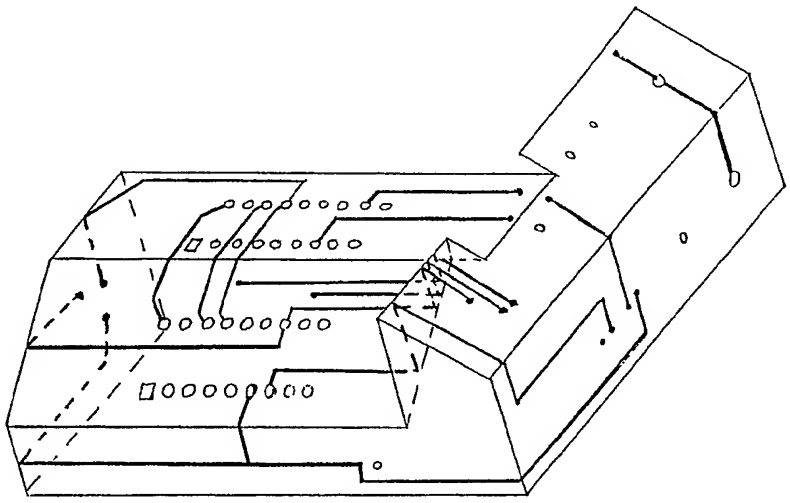

FIG. 4

EP 0 416 568 A2

METHOD OF DESIGNING THREE-DIMENSIONAL ELECTRICAL CIRCUITS

The present invention relates to the design of electrical circuits, and specifically to a method of producing electrical circuits on the surface of three-dimensional objects.

Many electrical or electronic circuits are made by means of printed circuit boards (PCBs) In which electrical conduction paths are defined as traces of copper on the surface of a flat sheet of insulating material. In mire complex versions, there may be multiple layers of circuit, for example,. on the two sides of the PCB, which are connected to each other by plated through holes or vias in the PCB.

PCBs provide both electrical connection and mechanical support for electronic components such as resistors, capacitors and integrated circuits. In addition, PCBs may include connectors used to couple the circuit, for example, to power sources, transducer or other circuits. Generally, PCBs are protected and supported in cases or other structures to make usable appliances.

In contradistinction to the flat, two-sided nature of the conventional PCB, a molded circuit board (MCB) is generally a three-dimensional object, having multiple surfaces. Thus, an MCB may combine both mechanical features and electrical functions into one artifact.

This combined functionality is not achieved without some cost, since the design of such an MCB combines the mechanical design technology of molded or solid parts with circuit layout methods used to make PCBs. While there are readily available computer software packages of program suites than can aid designers in either of these fields, these programs are generally incompatible and, thus, difficult to use in combination. This incompatibility is fundamental to the design methodologies and is not merely an issue of translating between different data formats.

The software packages for circuit design allow information about the placement of components in the PCB and the required connections among these components to be provided as an input data file, such as a netlist. The subsequent work of laying out the traces and vias is contained by the obligation to meet topological restrictions imposed by the input data file. Many of these program packages provide automatic trace routing algorithms which do some or all of the job with varying degrees of competence. All packages allow manual routing, in which traces and vias may be moved on the PCB while the circuit connections are preserved. Neither the automatic algorithms nor the ability to move traces ad vias is a part of the available methodology of conventional mechanical engineering design packages used to design three-dimensional objects. Conversely, the circuit layout and design packages do not admit of the existence of a third dimension, except in that they may provide for several parallel layers of circuit traces and the local connection between them.

The interrelationship between electrical and mechanical design s is very strong. Even in the design of flat circuit boards, the through holes which link circuit traces on opposite sides of the board and the holes which are used to support or attache components to the board are recognized as existing on all layers. Furthermore. many of these holes (i.e., vias) are logically associated with the traces.

When a designer adjusts a design in the electrical design package by moving the trace and, thus, causing a via to be moved, the computer software attempts to preserve the electrical-mechanical relationship by adjusting the routing on all layers of the board to conform to the new position of the via. Thus, in the electrical packages, a mechanical feature is automatically adapted to circuit requirements. In the mechanical engineering design packages, such topological relationships are not recognized. Mechanical features such as patterns of holes may be moved as a group, but in general, there is no way of relating these features to the continuity of adaptable decorative details on the surface of the object.

In attempting to extend circuit layout to surfaces in three dimensions, a new set of problems arise due to the limitations of the two-dimensional circuit layout tools. When a solid part is to have circuit traces drawn, for example, on its inner and outer surfaces, electrically identical traces on the two surfaces may be very different due to the thickness of the part near edges or corners, or due to features that break the surface on one side but are not replicated on the other side. Thus, features such as stiffening ribs, standoffs, outer walls and mounting structures may make the topologies of two sides of an object very different.

In the implementation of the design of a molded part, tools are generally used which carry, in mechanical form, the image of what would be the artwork (i.e., traces and connecting pads) in normal two-dimensional circuit processes. In some cases, these tools may be generated using normal circuit board processes, for example, when the part includes flat areas which can be dealt with as simple locally flat boards. However, to obtain full benefit of integrating mechanical and electrical designs, it is desirable to allow the traces to follow shaped surfaces. To do this, the tools which are used to form the part should impose the traces on the part.

However, these tools are themselves solid objects which ate fabricated by mechanical means such as machining or casting. Consequently, the traces and vias which constitute the electrical circuit should be specified in whatever system is used to fabricate the tools, be it mechanical drawings interpreted by a skilled machinist or input data files which are used to direct the motion of a numerically controlled machine tool.

The present invention is directed to a method for generating, from a representation of a two-dimensional circuit, a representation of a three-dimensional circuit which conforms to selected surfaces of a representation of a three-dimensional object, comprising the steps of

a) aligning one surface of the representation of the three-dimensional object in a coplanar relationship with the representation of the two-dimensional circuit;

b) establishing a correspondence between said one surface of the three-dimensional object and a portion of the representation of said two-dimensional circuit;

c) associating circuit features of said portion of the representation of the two-dimensional circuit with said one surface; and

d) copying said associated circuit features to the three-dimensional representation of said circuit.

Figure 1 is a perspective view of an exemplary three-dimensional object suitable for use with the present invention.

Figure 2 is plan drawing which illustrates the unfolding of the object of Fig. 1 to create a two-dimensional image of the object.

Figure 3 is a circuit trace diagram placed and routed onto the image shown in Fig. 2.

Figure 4 is a perspective view of the object shown in Fig. 1 with the circuit represented by the circuit trace diagram of Fig. 3 mapped onto its surface.

Figures 5 to 8 are flow-chart diagrams which are useful for describing the present invention.

The design process of an MCB begins with the creation of a three-dimensional model of the part. For the described embodiment of the invention, this three-dimensional model is generated in an Apollo workstation using the GMS CAD-CAM system of Unisys Corporation as the mechanical design package. In this system, a three-dimensional object is specified as a set of quadrilateral surfaces in three dimensions. These surfaces are also known as ruled surfaces. The mechanical package produces a data file describing the three-dimensional object. In the present embodiment of the invention, this file is in Initial Graphics Exchange Specification (IGES) format. An exemplary object is described below with reference to Fig. 1.

The next step in the process is to "unfold" this three-dimensional object to generate a two-dimensional map of the ruled surface which, in the final artifact, are to include circuitry. In the present invention, this step is performed by concatenating the ruled surface, generated using the GMS package, to produce an outline representing a two-dimensional PCB. As set forth below, although this outline should approximate the outline of the concatenated ruled surfaces, it is not necessarily isomorphic with the three-dimensional object. The process of generating an outline of this type is described bellow with reference to Fig. 2.

The two-dimensional outline is generated, from data describing the concatenated surfaces, in an electrical design package, such as the Visula system produced by Racal Redac. This design package is also used to specify the placement of features such as connecting pads and vias which define both electrical and mechanical coupling of circuit components. When these features have been placed, the electrical package is used to define a routing for traces that connect these pads and vias as indicated by the circuit description. For this embodiment of the invention, the placement is accomplished by an operator specifying the position of vias and the position and size of pads on the two-dimensional surface. The pads are identified as being associated with specific components or with specific networks in the netlist. For the exemplary electrical design package set forth above, the routing operation is performed automatically to link the connecting pads and vias as indicated by the netlist file. The electrical package produces a data file describing the placed and routed circuitry. In this embodiment of the invention, this file is a GERBER photoplotter file. Fig. 3 is an example of a circuit routing produced by the electrical package.

The GERBER file describes a connecting pad as a point on the two-dimensional surface and an aperture for a beam of light which is flashed on to produce an image of the pad on a photosensitive material. A trace is defined as a starting point, an ending point and an aperture for a beam of light which follows a path between the starting and ending points to produce an image of the trace.

In the next step, the data files representing the three-dimensional object and the two-dimensional circuit diagram are combined by a program, which also runs on the Apollo workstation, to generate a data file representing a three-dimensional object having a three-dimensional circuit imposed on its surface. This program combines the two files by using space transformation algorithms which "rotate" a mathematical representation of the object in space to align each surface with a portion of the circuit diagram. As each surface is aligned, the circuitry circumscribed by that surface is removed

from the data file for the circuit diagram and added to a data file for the three-dimensional circuit. In the exemplary embodiment of the invention, this program produces an IGES file which describes a mask that may be applied to the object generated in the first step to impose the circuitry on that object, Fig. 4 is an illustration of an exemplary mask.

In Fig. 1, the three-dimensional object shown in a perspective view includes seven surfaces: **101**, a hidden surface as shown In Fig. 1, that is bounded by vertices A, C, E, N and O; **102**, bounded by vertices A, B, J, G and C; **103**, bounded by vertices C, E, F and H; **104**, bounded by vertices E, N. M and F; **105**, bounded by vertices F, G, J, Q, P, M and L; **106**, bounded by vertices B, R, Q and J; and **107**, a hidden surface as shown in Fig. 1, bounded by vertices H, G, F and S.

As set forth above, this representation of the object may be generated using the mechanical design package. The object is generated one surface at a time, where each surface is a quadrilateral. To specify a surface, two line segments are described in terms of X, Y and Z coordinates and the endpoints of these segments are joined to form the quadrilateral. It is noted that some of the surfaces shown in Fig, 1 are actually composed of two or more quadrilaterals. For example, surfaces **102** and **105** are composed of two and three quadrilaterals, respectively.

The next step in the process is to generate a two-dimensional circuit layout area by concatenating the surfaces of the object shown in Fig. 1. This process resembles the generation of a two-dimensional map pf the three-dimensional object. This step is performed using the mechanical package. Each surface of the three-dimensional object is copied and references to a common plane. The various surfaces are then oriented so that adjacent surfaces are joined. However, some surfaces which are contiguous in the three-dimensional object, are separated as a result of the mapping process. An exemplary map is shown in Fig. 2.

A set of dimensions which describe the outer boundaries of the concatenated surface is applied as input values to the electrical package to create an equivalent two-dimensional map upon which the circuit is to be placed and routed. In order to link the map to the three-dimensional object, one vertex is selected as a common orienting point. In this exemplary map, the orienting point is the vertex O.

The map used by the electrical package does not need to be an accurate representation of every surface of the object. In Fig. 2, for example, the quadrilateral defined by the vertices D, J, G and H has been omitted from the map, and the surface **107**, defined by the vertices H, 5, F and G, has been incorporated in the map by lengthening the horizontal dimensions of the surfaces **103**, **104**, **105a**, **105b** and **105c**.

The surface **107** is handled in this manner so that the map may have a regular structure suitable for use by the electronics package. The inventor has determined that the routing achieved with this outline is more effective than that which may be achieved from a more isomorphically mapped outline. Since the width of surface **107** has been accounted for by lengthening the surfaces **103** and **105a**, any circuit laid out in the portion of the map near the junction of these two surfaces will be transferred onto the surface **107**.

Circuitry may or may not be transferred to the omitted surface defined by the vertices D, J, G and H. This depends on the order in which the surfaces are evaluated when the two-dimensional circuit is applied to the three-dimensional object. However, the portions of an object which are to be occupied by circuitry may be limited to selected adjacent surfaces by concatenating only those surfaces to generate the outline. A vertex of one of these surfaces should then be chosen as the orienting point of the outline.

For the purposes of the electrical package, the area onto which the circuit may be placed is represented by the outline of the map shown in Fig. 2, with the respective pairs of vertices C and C′ and H and G′ joined as indicated. The boundaries of the individual surfaces which were concatenated to generate this map are ignored by the electrical package.

The circuit produces by the electrical package for this example is shown in Fig. 3. This circuit was generated by defining connecting pads and vias corresponding to two integrated circuits **302** and **304** and a transistor **306** as well as various other pads and vias as indicated. In the electrical package, the individual vias are labeled to correspond to networks, or nets, of a circuit defined by a netlist data file. The electrical package then routes the circuit defined by this netlist onto the outline generated by the mapping process. Fig. 3 shows the result of this routing operation.

In the circuit diagram shown in Fig. 3, traces have been placed in the void region defined by the vertices N, E, C, C′, E′ and N′. If this were a flat board, traces such as these would generate an error indication from the electrical package. In this instance, however, the edge defined by the vertices N′, E′ and C′ is joined to the edge defined by the vertices N. E and C in the three-dimensional object and, thus, the void does not actually exist. Depending on the electrical package used, some error checking may need to be deimplemented to allow traces to be routed across voids of this type.

Care should be exercised where traces have been placed across voids, to ensure that the ends

of traces on opposite sides of the void are properly aligned for the three-dimensional object, For example, consider trace **308** generated by the electrical package. If the portions of this trace which extend across the void were omitted from the three-dimensional object, there would be no electrical connection because the end of the trace at the edge N'-E' would not meet the end of the corresponding trace at the edge N-E. To correct this problem, the operator of the electrical package moves trace **308** to the position **310** indicated in phantom. Other traces which extend across voids should be examined for similar adjustment.

If circuitry is to be placed on both sides of the three-dimensional object, each of the steps set forth above would be performed separately for the inner and outer surfaces of the object. In the present embodiment of the invention, the operator is responsible for ensuring that any differences in surface area due to curvature of the three-dimensional object or due to the thickness of its walls are properly represented by the generated circuit outlines.

Figures 5A and 5B are a flow-chart diagram which illustrates the overall process of creating a MCB for a three-dimensional part. Steps **502**, **504** and **506** relate to the design of the three-dimensional part, using the mechanical design package, and the creation of the IGES file describing the part. Steps **508** and **510** concern the creation of the two-dimensional surface outline from the three-dimensional part description. Steps **512**, **514** and **516** relate to processes performed using the electrical package. These steps in Figures 5A and 5B are described in detail above.

Step **518** of Fig. 5B is implemented, in the present embodiment of the invention, by a program written in the computer language SMALLTALK. A source listing of this program is included as an appendix to the present application. The step **518** is briefly described above and is described in detail below with reference to Figures 6A to 8. As set forth above, this step produces an output data file which describes the circuit in three-dimensional space as it has been mapped onto the three-dimensional part.

In step **520**, a test is made to determine if the output file is to be used to drive imaging equipment, in which case the file is sent to the imager at step **522**. Otherwise, at step **524**, the file is used by the mechanical design system, together with the IGES file that describes the three-dimensional part (provided as an input **526** to the step **524**), to generate, at step **528**, commands for a phototool. The phototool, in turn, generates a three-dimensional mask which may be used, as set forth above, to print the circuit on the surface of the three-dimensional part.

Figures 6A and 6B are a flow-chart diagram which provides grater detail on the process of mapping the two-dimensional circuit onto the three-dimensional object (step **518**).

The first step in this program, **602**, is to create an aperture list. This list is generated by the operator from data used to produce the GERBER circuit description file in the electrical package. This list defines the sizes of light beam apertures to be used by the photo-plotter to create images of the pads and traces on the photosensitive output medium.

In the next step, **604**, the operator creates a surface list, defining the sequence in which surfaces of the three-dimensional object are to be evaluated to receive circuitry from the two-dimensional circuit diagram. This list is created by the operator from a list of surface identifiers for the three-dimensional object. These identifiers are taken from the IGES file.

At step **606**, the program reads in the data describing the circuit from the GERBER data file **606**. This data is put into three-dimensional format at step **610**. In this step, each pad and each trace are defined as respective ruled surfaces, if a format compatible with the IGES file but specifying only two dimensions. The third dimension is provided later in the procedure. This step also generates separate TRACE and FLASH lists which include arrays of records which define the positions and dimensions of the respective traces and connecting pads in the two-dimensional circuit. A separate pair of lists is established for each surface of the three-dimensional object, however, at this time, all lists are empty except for the respective FLASH and TRACE lists that correspond to the first surface in the surface list. These two lists are initialized to include all of the respective flash and trace records for the two-dimensional circuit.

The next step, **612**, reads the mechanical surface data from the IGES part file 614. In step **616**, reached via the offpage connector 2, the three-dimensional circuit representation is mapped onto the three-dimensional part using the data from the IGES file. This step adds the third dimension to the circuit description that was generated at step **610**.

Step **618** illustrates the possible output data formats of the three-dimensional circuit description generated at step **616**. In the present embodiment of the invention, the output data is in the format of an IGES file. However, as illustrated in Fig. 6B, output data files may be prepared in several two-dimensional and three-dimensional formats.

Figures 7 and 8 are flow-chart diagrams which show details of the steps 616, described above. Fig. 7 is a flow-chart diagram of a procedure MAP, which is invoked at step **616**, and Fig. 8 is a flow-chart diagram of a procedure CLIP which is in-

voked from the procedure MAP as set forth below.

The procedures MAP and CLIP fit to two-dimensional circuit onto the surface of the three-dimensional object. The fitting process may be analoglzed to the wrapping of the three-dimensional object with a sheet of paper representing the circuit. As each surface of the object is aligned with the sheet of paper, all circuitry which covers the surface is clipped out of the paper and saved The clipped circuitry is marked as to the surface to which it belongs. This marking process establishes a dependency between portions of the circuit and respective surfaces of the three-dimensional object, That is to say the location in space of a section of the circuitry is dependent on the position in space of the surface onto which it has been mapped. Following this analogy, the folding and clipping continues until there is not more circuitry to be mapped or until there are no more surfaces to be mapped onto. When this process is applied to a three-dimensional object having both interior and exterior surfaces, each of these surfaces is mapped separately and the results of the separate mappings are merged as a final step.

The first step in the MAP procedure is to select a surface from the surface list. If this is the first invocation of the procedure, then first entry in surface list is selected. Otherwise, the selected surface is determined by the invocation of the MAP procedure as set forth below.

At step **704**, the selected surface is examined to determine if it has been used. If so, or if the TRACE and FLASH lists for the surface are empty, step **706** is executed to select the next surface. The FLASH and TRACE lists for the surfaces were established at step **610** above.

When a suitable surface has been selected, step **708** is executed to "rotate" and "translate" the selected surface into the X,Y plane. These operations are a mathematical translation of the representations of the three-dimensional object to achieve a surface representation having a normal vector which is parallel with the Z-axis. An exemplary method of performing the rotation is described in chapter 22 of a textbook by W.M. Newman et al, "Principles of Interactive Computer Graphics", McGraw Hill, 1979, pp. 333-354.

During the rotation and translation operations, the orienting point of the three-dimensional object is maintained as a reference point for the two-dimensional circuit diagram. Thus, when these operations are complete, there is a one-to-one correspondence between the target surface and a portion of the circuitry.

In step **710**, the portion of the circuitry which is aligned with the target surface is clipped from the data structure created in step 6120 and made dependent on the target surface. The clipping operation is described in detail below, in reference to Fig. 8.

In the next four steps, **712**, **714**, **716** and **718** the MAP procedure invokes itself recursively for each surface which is adjacent to the target surface, as indicated by the surface list. The steps **712** to **718** are executed if the surface list indicates that the target surface has respective left, right, top or bottom adjacent surfaces. These recursive invocations ensure that all surfaces of the three-dimensional object will be evaluated in the mapping process.

When all surfaces have been evaluated or when no remaining surface has entries in its TRACE or FLASH lists, step **720** terminates the mapping and clipping processes.

Figure 8 illustrates the clipping procedure invoked in step **710** above. The clipping algorithm used in this embodiment of the invention is generally the same as the Cohen-Sutherland algorithm which is set forth in chapter 6, pp. 65-67 of the above-referenced text by W.H. Newman et al.

In the first step, **802**, the left, right, top and bottom surfaces, if they exist, which are adjacent to the selected surface are identified. Next, at step **804**, a trace τ is selected from the TRACE list, (T[s]) for the selected surface s. At step **806**, the procedure determines if the trace τ is entirely within the clipping polygon defined by the selected surface. If so, a transformation matrix dependency for the trace τ is set, at step **808**, to indicate that the trace is mapped onto the surface s. Next, at step **810**, the trace τ is deleted from the TRACE list T[s] and placed in the master trace list.

If the trace τ is not entirely within the clipping polygon, it may either be entirely outside of, or partly inside and partly outside of the polygon. This determination is made at step **812**. If the trace is entirely outside the polygon, step **814** determines which edge of the polygon is closest to the trace and, at step **816**, assigns the trace to the TRACE list for the adjacent surface which shares that edge with the clipping polygon.

If the trace τ is only partly outside of the clipping polygon, the trace is clipped at the edge of the polygon to form two traces, τ' and τ'' and both traces are inserted into the TRACE list T[s] for the selected surface s.

If, after steps **810** or **816**, step **818** determines that the last trace has not been processed; or unconditionally after step **822**, step **823** is executed to select the next trace to be used as the trace τ. Step **823** branches to step **806** which processes the trace τ as set forth above.

If, at step **818**, it is determined that the TRACE list for the selected surface is empty, step **824** is executed which processes the FLASH lists for the surface in substantially the same manner as de-

scribed above. The processing of the FLASH lists is the same as the processing of the TRACE lists except that there is no analogue to steps **820** and **822** since, in this embodiment of the invention, a flash (connecting pad) may not be split across two surfaces. It is contemplated, however, that the system may be modified to accommodate such a split.

The final step, **826**, in the clipping procedure is to mark the selected surface s as used. This marking is tested in step **704** as set forth above.

It may desirable to specify that some ruled surfaces of a three-dimensional object are not to be imprinted with circuitry. This may be accomplished, for example, by marking these surfaces as used before the mapping and clipping processes are performed.

A method of producing a printed circuit on the surface of a three-dimensional object has been described. While this invention has been described in terms of a preferred embodiment, it is contemplated that it may be practiced as outlined above within the spirit and scope of the appended claims.

Claims

1. A method for generating, from a representation of a two-dimensional circuit, a representation of a three-dimensional circuit which conforms to selected surfaces of a representation of a three-dimensional object, comprising the steps of

   a) aligning one surface of the representation of the three-dimensional object in a coplanar relationship with the representation of the two-dimensional circuit;

   b) establishing a correspondence between said one surface of the three-dimensional object and a portion of the representation of said two-dimensional circuit;

   c) associating circuit features of said portion of the representation of the two-dimensional circuit with said one surface; and

   d) copying said associated circuit features to the three-dimensional representation of said circuit.

2. The method of claim 1 further including the step of repeating steps a) through d) for each selected surface in the representation of the three-dimensional object.

3. The method of claim 1 wherein step a) is preceded by a step of defining an order in which the selected surfaces are processed by steps a) through d).

4. The method of claim 1 further including the steps of

   e) deleting said associated circuit features from the representation of the two-dimensional circuit; and

   f) repeating steps a) through e) until no more circuit feature of the representation if the two-dimensional circuit can be aligned with the selected surfaces of the representation of the three-dimensional object.

5. The method of claim 1 wherein step a) is preceded by the steps of generating the representation of the three-dimensional object as a set of surfaces, including said selected surfaces, where each sur face in said set of surfaces is defined in terms of a shape and a position in space; mapping the representation of the three-dimensional object onto a two-dimensional map by concatenating the shapes of each of said selected surfaces; placing circuit features corresponding to components of a two-dimensional circuit on said two-dimensional map; and routing circuit features corresponding to connections of components of said two-dimensional circuit on said two-dimensional map to generate said representation of the two-dimensional circuit.

```
Model subclass: #Mcb
    instanceVariableNames: 'traces flashes geom range xrange yrange zrange transform slist mt mp '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Mcb comment:
'This class will model a 3 dimensional
molded circuit board and its traces.'!


!Mcb methodsFor: 'accessing'!

flashAdd: aPad
    "Add a pad to the collection of flashes"

    flashes add: aPad!

flashes
    "Return the collection of flashes"

    ↑ flashes!

flashes: aCollection
    "Save the collection of flashes"

    flashes ← aCollection!

geom
    "Return the collection of geometries"

    ↑ geom!

geomAdd: aGeom
    "Add geometry to the collection of geometries"

    geom ← aGeom!

mp
    "Return the collection of mapped flashes (pads)"

    ↑ mp!

mp: aCollection
    "Save the collection of mapped flashes (pads)"

    mp ← aCollection!

mt
    "Return the collection of mapped traces"

    ↑ mt!

mt: aCollection
    "Save the collection of mapped traces"
```

```
    mt ← aCollection!

range
    "return the 3D point which is the absolute extents of the model"

    ↑ range!

range: a3DPoint
    "save the 3D point which is the absolute extents of the model"

    range ← a3DPoint!

slist
    "Return the surface list which orders mapping"

    ↑ slist!

slist: aLinkedList
    "Save the linked list of surfaces"

    slist ← aLinkedList!

traceAdd: aTrace
    "Add a trace to the collection of traces"

    traces add: aTrace!

traces
    "Return the collection of traces"

    ↑ traces!

traces: aCollection
    "save the collection of traces"

    traces ← aCollection! !

!Mcb methodsFor: 'initialize'!

initialize
    "initialize a collection of traces and geometry"


transform ← OrderedCollection new: 1.
slist ← Array new: 10! !

!Mcb methodsFor: 'operations'!

change
    "modify a point on a trace"

    | aTrace aPoint bPoint|

    bPoint ← Point3D x:0.5 y:0.0 z:0.5.
    aTrace ← traces at: 1.
```

```
    aPoint ← aTrace beginPoint.
    aPoint ← aPoint + bPoint.
    aTrace beginPoint: aPoint.
    traces at: 1 put: aTrace.
    self changed!

changed: aString
    "pass along the Gerber command line being processed"


    self changed: aString!

getGerber
    "read in and process a Gerber file to
    initialize flashes and traces"

    | gerber |
    gerber ← Gerber new.
    gerber readFile.
    Transcript cr; show: ' Gerber file read complete ';cr;endEntry.
    self flashes: gerber flashes.
    self traces: gerber traces.
    self range: gerber extents!

getIges
    "read in and process a Iges file to
    initialize surfaces"

    | iges |
    iges ← Iges new.
    iges readFile.
    Transcript cr; show: ' Iges file read complete ';cr;endEntry.
    self geomAdd: iges surfaces!

getSlist
    "get list of surfaces to process"

    | cnt link slink aString aFileName aFile |
    cnt ← 1.
    link ← LinkedList new.
    slink ← SurfList new.
    (BinaryChoice
        message: 'Do you wish to read in a surface list?')
        ifTrue:[
                aFileName ← FillInTheBlank
                    request:'What is the name of the surface file?'
                    initialAnswer:'surface.list'.
                aFile ← FileStream oldFileNamed: aFileName.
                slist ← (aFile fileIn).
                aFile close.]
        ifFalse:[
    [ BinaryChoice
        message: 'Create a surface list?']
    whileTrue:[  [BinaryChoice
                    message: 'Add a surface?']
                whileTrue: [ aString ← FillInTheBlank
```

```
                                request: 'Enter surface name.'
                                initialAnswer: 'S1'.
                            slink name: aString.
                            link add: (slink deepCopy).
                            slink release.
                            aString release.].
                    self slist at: cnt put: (link deepCopy).
                    [link size > 0]
                    whileTrue:[link removeFirst].
                    cnt ← cnt + 1.].
    (BinaryChoice
        message: 'Do you wish to save the surface list?')
        ifTrue:[
    aFileName release.
    aFileName ← FillInTheBlank
                    request: 'Enter surface list file name for saving.'
                    initialAnswer: 'surface.list'.
    aFile ← FileStream fileNamed: aFileName.
    aFile reset.
    slist storeOn: aFile.
    aFile close.].
    ].
    Transcript cr; show: ' Surface list creation complete. ';cr;endEntry.!

inspect
    "inspect the traces"

    traces inspect!

mapping
    "map circuits onto surfaces"

    | aSurface mappedCircuits surfName|
    surfName ← (self slist at: 1) first.
    1 to: (geom size) do:[:cnt|
    ((self geom at: cnt) name = (surfName name)) ifTrue:[
    aSurface ← self geom at: cnt.].].
    mappedCircuits ← aSurface mapToNextSurface: self traces with: self flashes and: self geom and: self slist.
    self mt: (mappedCircuits at: 1).
    self mp: (mappedCircuits at: 2).
    Transcript cr;show:'Mapping Complete';cr;endEntry! !
"-- -- -- -
```

↑newMcb! !

```
View subclass: #McbView
    instanceVariableNames: 'volume dorigin projection '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
McbView comment:
'This class is for viesing a 3 dimensional
molded circuit board.'!


!McbView methodsFor: 'displaying'!

displayPad
    "Display the model's Pads"

    | set1 aPad tVal aForm width dPad nPad hw|

    ( self model mp = nil) ifFalse:[ set1 ← self model mp.]
        ifTrue:[set1 ← self model flashes.].
    self dorigin: self insetDisplayBox bottomLeft. "set origin to lower left corner of window"
    tVal ← set1 size.
    1 to: tVal do: [ :index |
        aPad ← set1 at: index.
        nPad ← aPad convToView: self volume. "convert world to display coordinates"
        dPad ← self projectPad: (self projection) with: nPad.   "do projection calculations here"
        dPad ← self scale: dPad.   "scale the object to fit the window"
        width ← self findPadWidth: aPad.   "set shape correctly for display"
        hw ← (width@width) / (2@2). " offset pad center 1/2 width"
        dPad ← dPad - hw.
        (aPad padshape = 'round') ifTrue:[ "if pad is round set size"
            aForm ← Form dotOfSize: width.
            aForm offset: dPad.]
        ifFalse:[
            aForm ← Form new extent: width@width offset: dPad.          "if pad is square set width"
            aForm black.].                                  "turn it black"
        aForm displayOn: Display
            at: self dorigin
            clippingBox: self insetDisplayBox.]!

displaySurface
    "Display the model's surfaces"

    | set1 aSurface tVal aForm width cSurface bSurface hw|

    set1 ← self model geom.
    self dorigin: self insetDisplayBox bottomLeft. "set origin to lower left corner of window"
    tVal ← set1 size.
    1 to: tVal do: [ :index |
        aSurface ← set1 at: index.
        bSurface ← aSurface convToView: self volume. "convert world to display coordinates"
        "bSurface inspect."
        cSurface ← self projectSurface: (self projection) with: bSurface.    "do projection calculations here"
        "cSurface inspect."
        cSurface ← self scale: cSurface.   "scale the object to fit the window"
        aForm ← Form new extent: 1@1.          "make a form which could vary by width"
```

```
        aForm black.                                    "turn it black"
        cSurface form: aForm.                                   "use the black form for display"
        cSurface displayOn: Display
            at: self dorigin
            clippingBox: self insetDisplayBox.]!

displayTrace
    "Display the model's traces"

    | set1 aTrace tVal aForm width aLine nTrace hw|

    ( self model mt = nil ) ifFalse:[ set1 ← self model mt.]
        ifTrue:[set1 ← self model traces.].
    self dorigin: self insetDisplayBox bottomLeft.  "set origin to lower left corner of window"
    tVal ← set1 size.
    1 to: tVal do: [ :index |
        aTrace ← set1 at: index.
        nTrace ← aTrace convToView: self volume. "convert world to display coordinates"
        aLine ← self projectTrace: (self projection) with: nTrace.   "do projection calculations here"
        aLine ← self scale: aLine.   "scale the object to fit the window"
        width ← self findTraceWidth: aTrace.   "set width correctly for display"
        hw ← (width@width) / (2@2). "offset line points for form width"
        aLine beginPoint: (aLine beginPoint)- hw.
        aLine endPoint: (aLine endPoint)-hw.
        aForm ← Form new extent: width@width.          "make a form vary by width"
        aForm black.                                   "turn it black"
        aLine form: aForm.                                  "use the black form for display"
        aLine displayOn: Display
            at: self dorigin
            clippingBox: self insetDisplayBox.]!

displayView
    "Display the model's pads and traces"

    self volume: self model range.
    self displaySurface.
    self displayPad.
    self displayTrace!

redraw
    "redisplay upon command from controller"

    self display!

update
    "redisplay upon change in model"

    self display!

update: aParameter
    "Display the value of the model."

    | box pos |
    self clearInside.
        "Position the text at the left side of the display area."
    box ← self insetDisplayBox.          "get the view's box"
```

```
    pos ← box origin + (4 @ (box extent y / 3)).
    "Concatenate the components of the output string and display them."
    ('command: ', aParameter) asDisplayText displayAt: pos! !

!McbView methodsFor: 'initialization'!

startup
    "set view variables to default values"

    self volume: 12@12@12.
    self dorigin: 0@0@0.
    self projection: 1! !

!McbView methodsFor: 'transformation'!

findPadWidth: aPad
    "set width for form so object is displayed at correct size"

    | tx wt nw |
    tx ← self volume x.
    wt ← aPad padsize.
    nw ← ((wt / tx) * 1000) truncated.
    nw < 1 ifTrue:[ nw ← 1.].
    ↑nw!

findTraceWidth: aTrace
    "set width for form so object is displayed at correct size"

    | tx wt nw |
    tx ← self volume x.
    wt ← aTrace w.
    nw ← ((wt / tx) * 1000) truncated.
    nw < 1 ifTrue:[ nw ← 1.].
    ↑nw!

projectPad: anInteger with: aPad
    "cabinet projection for now. In future will control how the drawing is projected
    by changing the coordinates to match the required view"

    | nPad mat |

    mat ← Matrix new: 4 by: 4.
    mat setToZero.
    mat atPoint: 1@1 put: 1.0.
    mat atPoint: 2@2 put: 1.0.
    mat atPoint: 4@4 put: 1.0.
    mat atPoint: 1@3 put: 0.4477.
    mat atPoint: 2@3 put: 0.8941.
    aPad rotate: mat.
    nPad ← aPad coordinates as2DPoint.
    ↑nPad!

projectSurface: anInteger with: aSurface
    "dummy for now. In future will control how the drawing is projected
    by changing the coordinates to match the required view"
```

```
    | mat |

    mat ← Matrix new: 4 by: 4.
    mat setToZero.
    mat atPoint: 1@1 put: 1.0.
    mat atPoint: 2@2 put: 1.0.
    mat atPoint: 4@4 put: 1.0.
    mat atPoint: 1@3 put: 0.4471.
    mat atPoint: 2@3 put: 0.8941.
    aSurface rotate: mat.
    aSurface e1: (aSurface e1 as2DLine).
    aSurface e2: (aSurface e2 as2DLine).
    aSurface e3: (aSurface e3 as2DLine).
    aSurface e4: (aSurface e4 as2DLine).
    ↑ aSurface!

projectTrace: anInteger with: a3Dline
    "cabinet projection for now. In future will control how the drawing is projected
    by changing the coordinates to match the required view"

    | mat |

    mat ← Matrix new: 4 by: 4.
    mat setToZero.
    mat atPoint: 1@1 put: 1.0.
    mat atPoint: 2@2 put: 1.0.
    mat atPoint: 4@4 put: 1.0.
    mat atPoint: 1@3 put: 0.4471.
    mat atPoint: 2@3 put: 0.8941.
    a3Dline rotate: mat.
    ↑a3Dline as2DLine!

scale: anObject
    "scale an object. It must understand the scaleBy: message"

    ↑ anObject scaleBy: (self displayTransformation scale)! !

!McbView methodsFor: 'accessing'!

dorigin
    "return the x,y,and z drawing origin of the display space in inches as a 3D point."

    ↑dorigin!

dorigin: aPoint
    "set the x,y,and z drawing origin of the display space in inches as a 3D point."

    dorigin ← aPoint!

projection
    "return the projection selector of the display space to be used in selecting function"

    ↑projection!

projection: anInteger
    "return the projection selector of the display space to be used in selecting function"
```

```
    projection ← anInteger!

volume
    "return the x,y,and z volume of the display space in inches as a 3D point."

    ↑ volume!

volume: aPoint
    "set the x,y,and z volume of the display space in inches as a 3D point."

    volume ← aPoint! !
"-- -- -- -
```

```
MouseMenuController subclass: #McbController
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
McbController comment:
'This class is the controller for constructing a 3 dimensional
molded circuit board.'!


!McbController methodsFor: 'menu messages'!

change
    "tell model to change"

    self model change!

gerber
    "tell model to read in a Gerber file."

    self model getGerber!

iges
    "tell model to read in an Iges file."

    self model getIges!

inspect
    "tell model to inspect traces"

    self model inspect!

inspectview
    "inspect the view"

    self view perform: #inspect!

mapping
    "tell model to map circuits."

    self model mapping!

redraw
    "tell view to redisplay"

    self view redraw!

slist
    "tell model to read in an surface list file."

    self model getSlist! !

!McbController methodsFor: 'control defaults'!

isControlActive
```

```
    "Inherits control from superclass MenuMouseController
    As the blue button menu is used for reframing views only,
    mcb is activated by pressing the other two buttons"

    ↑ super isControlActive & sensor blueButtonPressed not! !

!McbController methodsFor: 'initialize'!

initialize
    "McbController initialization"

    super initialize.
    self yellowButtonMenu: (PopUpMenu labels:
        'Redraw
Change
Inspect
Inspect View
Gerber In
IGES In
Surface List
Mapping')
    yellowButtonMessages:#(redraw change inspect inspectview gerber iges slist mapping)! !
```

```
Object subclass: #Gerber
    instanceVariableNames: 'g d x1 x2 y1 y2 oldApt size m apShape traces flashes apertures xmin xmax ymin
ymax zmin zmax '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Gerber comment:
'This class processes Gerber photoplotter
files and reconstructs the circuitry for
use by class Mcb.'!


!Gerber methodsFor: 'parsing'!

getD: aString
    "extract the D command from the command line
    and save it in d"

    | pString s |
    s ← aString size.
    1 to: s do: [:index |
        (SD = (aString at: index) )
            ifTrue:[pString ← String with:(aString at: index) with: (aString at: index + 1)
                    with: (aString at: index + 2).
                    self d: pString.].]!

getG: aString
    "extract the G command from the command line
    and save it in g"

    | pString s |
    s ← aString size.
    1 to: s do: [:index |
        ($G = (aString at: index) )
            ifTrue:[pString ← String with:(aString at: index) with: (aString at: index + 1)
                    with: (aString at: index + 2).
                    self g: pString.].].
    "self inspect."!

getM: aString
    "extract the M command from the command line
    and save it in m"

    | pString s |
    s ← aString size.
    1 to: s do: [:index |
        (SM = (aString at: index) )
            ifTrue:[pString ← String with:(aString at: index) with: (aString at: index + 1)
                    with: (aString at: index + 2).
                    self m: pString.].]!

getX: aString
    "extract the X value from the command line
    and save it in x2, as the current x value.
    This method assumes the Gerber numbers
```

```
    are in 2.3 format, with or without leading zeros."

    | aFloat aNumber s aChar i |
    aNumber ← 0.
    s ← aString size.
    1 to: s do: [:index |
        (SX = (aString at: index) )
            ifTrue:[ i← index +1.
                [(aChar ← aString at: i) isDigit] whileTrue: [
                    aNumber ← aNumber *10 + (aChar asInteger - 48).
                    i ← i+1.].
                aFloat ← aNumber / 1000.0.
                self x2: aFloat.
                ( aFloat < xmin) ifTrue:[ self xmin: aFloat.].
                ( aFloat > xmax) ifTrue:[ self xmax: aFloat.].
                ].].!

getY: aString
    "extract the Y value from the command line
    and save it in y2, as the current y value.
    This method assumes the Gerber numbers
    are in 2.3 format, with or without leading zeros."

    | aFloat aNumber s aChar i |
    aNumber ← 0.
    s ← aString size.
    1 to: s do: [:index |
        (SY = (aString at: index) )
            ifTrue:[ i← index +1.
                [(aChar ← aString at: i) isDigit] whileTrue: [
                    aNumber ← aNumber *10 + (aChar asInteger - 48).
                    i ← i+1.].
                aFloat ← aNumber / 1000.0.
                self y2: aFloat.
                ( aFloat < ymin) ifTrue:[ self ymin: aFloat.].
                ( aFloat > ymax) ifTrue:[ self ymax: aFloat.].
                ].]!

parse: aCommand
    "Take a command line and parse it into its proper components."

    self getG: aCommand.
    self getX: aCommand.
    self getY: aCommand.
    self getD: aCommand.
    self getM: aCommand! !

!Gerber methodsFor: 'accessing'!

aperture
    "return the string which is the D aperture"

    ↑aperture!

aperture: aString
    "save the string which is the D aperture"
```

```
    aperture ← aString!

apShape
    "return the string which is the aperture shape"

    ↑apShape!

apShape: aString
    "save the string which is the aperture shape"

    apShape ← aString!

d
    "return the string which is the D command"

    ↑d!

d: aString
    "save the string which is the D command"

    d ← aString!

flashAdd: aPad
    "Add a pad to the collection of flashes"

    flashes add: aPad!

flashes
    "Return the collection of flashes"

    ↑ flashes!

flashes: aCollection
    "Save the collection of flashes"

    flashes ← aCollection!

g
    "return the string which is the G command"

    ↑g!

g: aString
    "save the string which is the G command"

    g ← aString!

m
    "return the string which is the M command"

    ↑m!

m: aString
    "save the string which is the M command"
```

```
    m ← aString!

size
    "return the number which is the aperture size"

    ↑size!

size: aFloat
    "save the number which is the aperture size"

    size ← aFloat!

traceAdd: aTrace
    "Add a trace to the collection of traces"

    traces add: aTrace!

traces
    "Return the collection of traces"

    ↑ traces!

traces: aCollection
    "save the collection of traces"

    traces ← aCollection!

x1
    "return the number which is the previous x location"

    ↑x1!

x1: aFloat
    "save the number which is the previous x location"

    x1 ← aFloat!

x2
    "return the number which is the current x location"

    ↑x2!

x2: aFloat
    "save the number which is the current x location"

    x2 ← aFloat!

xmax
    "return the number which is the maximum x location"

    ↑xmax!

xmax: aFloat
    "save the number which is the maximum x location"

    xmax ← aFloat!
```

```
xmin
    "return the number which is the minimum x location"

    ↑xmin!

xmin: aFloat
    "save the number which is the minimum x location"

    xmin ← aFloat!

y1
    "return the number which is the previous y location"

    ↑y1!

y1: aFloat
    "save the number which is the previous y location"

    y1 ← aFloat!

y2
    "return the number which is the current y location"

    ↑y2!

y2: aFloat
    "save the number which is the current y location"

    y2 ← aFloat!

ymax
    "return the number which is the maximum y location"

    ↑ymax!

ymax: aFloat
    "save the number which is the maximum y location"

    ymax ← aFloat!

ymin
    "return the number which is the minimum y location"

    ↑ymin!

ymin: aFloat
    "save the number which is the minimum y location"

    ymin ← aFloat!

zmax
    "return the number which is the maximum z location"

    ↑zmax!
```

```
zmax: aFloat
    "save the number which is the maximum z location"

    zmax ← aFloat!

zmin
    "return the number which is the minimum z location"

    ↑zmin!

zmin: aFloat
    "save the number which is the minimum z location"

    zmin ← aFloat! !

!Gerber methodsFor: 'private'!

execute
    "process the parsed Gerber command line"


    ('M02' = self m) ifFalse: [  "not the end of file"
        ('G54' = self g) ifTrue: [  "change the aperture"
            oldApt ← self d.
            apShape ← (apertures at: self d) shape.
            size ← (apertures at: self d) size.]
            ifFalse: [ "create the plot entity"
                ('D01' = self d) ifTrue: [ "draw a trace"
                    self makeTrace.].
                ('D02' = self d) ifTrue: [ " move to new location"
                    self x1: self x2.
                    self y1: self y2.].
                ('D03' = self d) ifTrue: [ "create a pad"
                    self makePad.].
            ].
        ]!

extents
    "return the absolute extents of the coordinates"

    | aPoint x y z|

    z ← zmax.
    x ←(xmax + 1.0).
    y ←(ymax + 1.0).
    ↑aPoint ← x@y@z!

makePad
    "create a pad and put it in the data base"

    | z aPoint aPad |
    aPoint ← x2@y2@0.0.
    aPad ← Pad new.
    aPad coordinates: aPoint.
    aPad padshape: self apShape.
    aPad padsize: self size.
```

```
    self flashAdd: aPad.
    self x1: x2.
    self y1: y2!

makeTrace
    "create a trace and put it in the data base"

    | z aPoint bPoint aTrace |
    bPoint ← x2@y2@0.0.
    aPoint ← x1@y1@0.0.
    aTrace ← Trace new.
    aTrace beginPoint: aPoint.
    aTrace endPoint: bPoint.
    aTrace w: self size.
    self traceAdd: aTrace.
    self x1: x2.
    self y1: y2!

readFile
    "Read and process a Gerber file.
    This is the main method which kicks off the rest of the
    Gerber methods"

    | gerberFile commandLine aFileName aFile theFileName|

    theFileName ← FillInTheBlank
        request:'What is the name of the aperture file?'
        initialAnswer:'aperture.dict'.
    aFile ← FileStream oldFileNamed: theFileName.
    apertures ← (aFile fileIn). "apertures becomes the dictionary containing all aperture information"
    aFile close.
    self x1: 0.0.  "set all coordinates to zero"
    self y1: 0.0.
    self x2: 0.0.
    self y2: 0.0.
    self m: 'M00'.
    self xmin: 0.0.
    self xmax: 0.0.
    self ymin: 0.0.
    self ymax: 0.0.
    self zmin: 0.0.
    self zmax: 12.0.

    traces ← OrderedCollection new: 2.
    flashes ← OrderedCollection new:3.

    aFileName ← FillInTheBlank
        request: 'What is the Gerber file name?'
        initialAnswer: 'gecomp.gbr'.
    gerberFile ← (FileStream oldFileNamed: aFileName) readOnly.
    gerberFile reset.
    [gerberFile atEnd] whileFalse:[
    commandLine ← String readFile: gerberFile.
    Transcript cr; show: commandLine; cr; endEntry.
    self parse: commandLine.
    "self inspect."
```

```
self execute.
"self inspect."
commandLine release.].
gerberFile close.! !
```

```
Object subclass: #Aperture
    instanceVariableNames: 'name shape size '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Aperture comment:
'This class is an aperture for a Gerber photoplotter'!


!Aperture methodsFor: 'comparing'!

<= anAperture
    "compare the name of the apertures
    for an ASCII sort"

    ↑ self name <= anAperture name! !

!Aperture methodsFor: 'accessing'!

name
    "return the string that is the Gerber D
    number for this aperture"

    ↑ name!

name: aString
    "save the string that is the Gerber D
    number for this aperture"

    name ← aString!

shape
    "return the string that is the shape
     for this aperture"

    ↑ shape!

shape: aString
    "save the string that is the shape
     for this aperture"

    shape ← aString!

size
    "return the number that is the size
     for this aperture in inches"

    ↑ size!

size: aFloat
    "save the number that is the size
     for this aperture in inches"

    size ← aFloat! !
```

```
Line3D subclass: #Trace
    instanceVariableNames: 'w surface '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Trace comment:
'This class is the structure for
printed circuit traces.
Traces are 3D lines with width.
They are 3D because of the
MCB application.'!


!Trace methodsFor: 'accessing'!

surface
    "Return the surface associated with the circuit trace"

    ↑ surface!

surface: aString
    "Set the surface associated with the circuit trace"

    ↑ surface ← aString!

w
    "Return the line width of the circuit trace"

    ↑ w!

w: value
    "Set the line width of the circuit trace"

    ↑ w ← value! !

!Trace methodsFor: 'transformation'!

convToView: a3DPoint
    "convert world coordinates to view coordinates based on world extents
    a3DPoint is the extents of the world view."

    | aTrace point1 point2 nTrace wPoint |

    wPoint ← 1024.0@800.0@800.0.
    point1 ← self beginPoint. "begin translation from world to display coordinates"
    point2 ← self endPoint.
    point1 ← (point1 / a3DPoint) * wPoint. " constant is view maximums"
    point2 ← (point2 / a3DPoint) * wPoint. "assume z extent in view same as y"
    point1 y: ( 0.0 - (point1 y)). "invert y to move display origin to lower left"
    point2 y: ( 0.0 - (point2 y)).
    nTrace ← Trace new.
    nTrace beginPoint: point1.
    nTrace endPoint: point2.
    "nTrace inspect."
    ↑ nTrace!
```

```
mapToSphere: a3DPoint
    "map circuit to sphere centered in the model volume."

    | aTrace point1 point2 nTrace a b c d t ph x1 y1 z1 r |

    point1 ← self beginPoint. "begin translation from world to display coordinates"
    point2 ← self endPoint.
    a ← (Float pi).
    b ← (-1.0*(a/2.0)).
    c ← b.
    d ← (3.0*a)/4.0.
    a3DPoint z: 1.0. "we don't want to change z."
    r ← (a3DPoint x) /4.0.
    point1 ← (point1 / a3DPoint). " set coordinates to fractional part of volume."
    point2 ← (point2 / a3DPoint).
    x1 ← point1 x.
    y1 ← point1 y.
    z1 ← point1 z.
    t ← (a*x1)+b.
    ph ← (c*y1)+d.
    x1 ← (r*(t sin)*(ph sin))+((a3DPoint x)/2.0).
    y1 ← (r*(ph cos))+((a3DPoint y)/2.0).
    z1 ← (r*(t cos)*(ph sin))+((a3DPoint z)/2.0).
    z1 ← z1 + point1 z.
    point1 x: x1.
    point1 y: y1.
    point1 z: z1.
    x1 ← point2 x.
    y1 ← point2 y.
    z1 ← point2 z.
    t ← (a*x1)+b.
    ph ← (c*y1)+d.
    x1 ← (r*(t sin)*(ph sin))+((a3DPoint x)/2.0).
    y1 ← (r*(ph cos))+((a3DPoint y)/2.0).
    z1 ← (r*(t cos)*(ph sin))+((a3DPoint z)/2.0).
    z1 ← z1 + point2 z.
    point2 x: x1.
    point2 y: y1.
    point2 z: z1.
    point1 ← point1 * a3DPoint.
    point2 ← point2 * a3DPoint.
    point1 x: (point1 x +(a3DPoint x / 2.0)). "translate from origin back to center of volume"
    point1 y: (point1 y+(a3DPoint y / 2.0)).
    point2 x: (point2 x +(a3DPoint x / 2.0)). "translate from origin back to center of volume"
    point2 y: (point2 y+(a3DPoint y / 2.0)).
    self beginPoint: point1.
    self endPoint: point2.
    "self inspect."!

rotate: aMatrix
    "rotate trace as defined by transformation matrix"

    | m1 m2 |
    m1 ← self beginPoint asMatrix.
    m2 ← self endPoint asMatrix.
```

```
m1 ← m1 * aMatrix.
m2 ← m2 * aMatrix.
self beginPoint: m1 asPoint3D.
self endPoint: m2 asPoint3D! !
```

```
DisplayObject subclass: #Pad
    instanceVariableNames: 'padshape padsize coordinates surface '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Pad comment:
'This class is the structure for
constructing printed circuit pads.
At this time pads are restricted
to the shapes round and square.'!


!Pad methodsFor: 'accessing'!

coordinates
    "Answer the center point on the receiver."

    ↑coordinates!

coordinates: a3DPoint
    "Answer the center point on the receiver."

    coordinates ← a3DPoint!

padshape
    "return the string 'circle' or 'square'"

    ↑padshape!

padshape: aString
    "set to the string 'circle' or 'square'"

    padshape ← aString!

padsize
    "return the size in inches,
    either a diameter or side."

    ↑padsize!

padsize: aNumber
    "set the size in inches,
    either a diameter or side."

    padsize ← aNumber!

surface
    "Answer the surface associated with the receiver."

    ↑surface!

surface: aString
    "Save the surface associated with the receiver."

    surface ← aString! !
```

```
!Pad methodsFor: 'transformation'!

convToView: a3DPoint
    "convert world coordinates to view coordinates based on world extents
    a3DPoint is the extents of the world view."

    | aPad point1 point2 nPad wPoint |

    wPoint ← 1024.0@800.0@800.0.
    point1 ← self coordinates. "begin translation from world to display coordinates"
    point1 ← (point1 / a3DPoint) * wPoint. " constant is view maximums"
                                                "assume z extent in view same as y"
    point1 y: ( 0.0 - (point1 y)). "invert y to move display origin to lower left"
    nPad ← Pad new.
    nPad coordinates: point1.
    "nPad inspect."
    ↑ nPad!

mapToSphere: a3DPoint
    "convert world coordinates to view coordinates based on world extents
    a3DPoint is the extents of the world view."

    | aPad point1 point2 nPad a b c d t ph x1 y1 z1 r |

    point1 ← self coordinates. "begin translation from world to display coordinates"
    a ← (Float pi).
    b ← (-1.0*(a/2.0)).
    c ← b.
    d ← (3.0*a)/4.0.
    a3DPoint z: 1.0. "we don't want to change z."
    point1 ← (point1 / a3DPoint). " constant is view maximums"
                                                "assume z extent in view same as y"
    r ← (a3DPoint x) /4.0.
    x1 ← point1 x.
    y1 ← point1 y.
    z1 ← point1 z.
    t ← (a*x1)+b.
    ph ← (c*y1)+d.
    x1 ← (r*(t sin)*(ph sin))+((a3DPoint x)/2.0).
    y1 ← (r*(ph cos))+((a3DPoint y)/2.0).
    z1 ← (r*(t cos)*(ph sin))+((a3DPoint z)/2.0).
    z1 ← z1 + point1 z.
    point1 x: x1.
    point1 y: y1.
    point1 z: z1.
    self coordinates: point1.
    "self inspect."!

rotate: aMatrix
    "rotate pad as determined by transformation matrix"

    | m1 |
    m1 ← self coordinates asMatrix.
    m1 ← m1 * aMatrix.
    self coordinates: m1 asPoint3D! !
```

```
!Pad methodsFor: 'initialize-release'!

coordinates: a3DPoint padshape: aString padsize: aFloat
    "set the variables for a pad"

    self coordinates: a3DPoint.
    self padshape: aString.
    self padsize: aFloat! !
```

```
Object subclass: #Iges
    instanceVariableNames: 'dict param surfaces '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Iges comment:
'This class reads, writes, and processes IGES files.'!


!Iges methodsFor: 'private'!

readFile
    "Read and process an IGES file.
    This is the main method which kicks off the rest of the
    IGES methods"

    | igesFile commandLine commandLine2 aFileName aFile deNumber aString index |

    aFileName ← FillInTheBlank
        request: 'What is the IGES file name?'
        initialAnswer: 'surface.iges'.
    igesFile ← (FileStream oldFileNamed: aFileName) readOnly.
    igesFile reset.
    [igesFile atEnd] whileFalse:[
    commandLine ← String readRecord: igesFile.
    Transcript cr; show:'size ';print:(commandLine size);show: ' type chr ';print: (commandLine at:
73);cr;endEntry.
    Transcript cr; show: commandLine; cr; endEntry.
     (commandLine at: 73) == $D
    ifTrue:[
            commandLine2 ← String readRecord: igesFile.
    Transcript cr; show:'size ';print:(commandLine size);show: ' type chr ';print: (commandLine at:
73);cr;endEntry.
            self getD: commandLine with: commandLine2]
    ifFalse:[(commandLine at: 73) == $P
            ifTrue:[
                    index ← 65.
                    [(commandLine at: index) == $ ]
                    whileTrue:[index ← index + 1].
                    deNumber ← ((commandLine copyFrom: index to: 72) asNumber).
                    self getP: commandLine with: deNumber]].
    commandLine release.].
    igesFile close.! !

!Iges methodsFor: 'accessing'!

dict
    "Return the dictionary which is the directory portion of the IGES file"

    ↑ dict!

dict: aDictionary
    "Create the dictionary which is the directory portion of the IGES file"

    ↑ dict ← aDictionary!
```

```
param
    "Return the collection which is the parameter geometry portion of the IGES file"

    ↑ param!

param: aCollection
    "Create the collection which is the parameter portion of the IGES file"

    ↑ param ← aCollection!

surfaces
    "Return the surfaces which are created from the IGES file"

    ↑ surfaces!

surfaces: aCollection
    "Create the collection which will be the surfaces created from the IGES file"

    ↑ surfaces ← aCollection! !

!Iges methodsFor: 'parsing'!

getD: aString with: aString2
    "process the directory entry into dictionary keyed by directory number"

    | dir newString |
    dir ← Directory new.
    dir eType: ((newString ← self getField: aString at: 1) asNumber).
    dir paramDataPntr: ((newString ← self getField: aString at: 9) asNumber).
    dir refByOther: ((newString ← self getField: aString at: 65) asNumber).
    dir dirNumber: ((newString ← self getField: aString at: 74) asNumber).
    dir paramRecCnt: ((newString ← self getField: aString2 at: 25) asNumber).
    dir eLabel: (newString ← self getField: aString2 at: 57).
    "dir inspect."
    self dict at: (dir dirNumber) put: dir!

getField: aString at: aNumber
    "get the required field from the command line string"

    | stop newString index |

    index ← aNumber.
    (aString size - aNumber) < 7
        ifTrue: [stop ← (aString size - aNumber)]
        ifFalse: [stop ← 7].
    [(aString at: index) == $ ]
    whileTrue:[ index ← index + 1 ].
    index > (aNumber + stop)
    ifTrue:[ newString ← '0']
    ifFalse:[
        newString ← aString copyFrom: index to: (aNumber + stop).].
    "newString inspect."
    ↑ newString!

getLine: aString
```

```
    "process a parameter command for a line"

    | x1 x2 y1 y2 z1 z2 type stop index aLine newString |

    index ← 1.
    stop ← aString indexOfSubCollection: ',' startingAt: index.
    type ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
    index ← stop + 1.
    newString release.
    type = 110
    ifTrue: [
        stop ← aString indexOfSubCollection: ',' startingAt: index.
        x1 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        index ← stop + 1.
        newString release.
        stop ← aString indexOfSubCollection: ',' startingAt: index.
        y1 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        index ← stop + 1.
        newString release.
        stop ← aString indexOfSubCollection: ',' startingAt: index.
        z1 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        index ← stop + 1.
        newString release.
        stop ← aString indexOfSubCollection: ',' startingAt: index.
        x2 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        index ← stop + 1.
        newString release.
        stop ← aString indexOfSubCollection: ',' startingAt: index.
        y2 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        index ← stop + 1.
        newString release.
        stop ← aString indexOfSubCollection: ';' startingAt: index.
        z2 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        aLine ← Line3D new.
        aLine beginPoint: x1@y1@z1.
        aLine endPoint: x2@y2@z2.
        ↑ aLine]
    ifFalse:[ Transcript cr; show: 'Not a Line'; cr;
                show: 'Entity type = '; print: type;cr; endEntry.]!

getP: aString with: dirNumber
    "process the parameter file for lines or surfaces"

    | index stop type newString aLine aSurface |

    index ← 1.
    stop ← aString indexOfSubCollection: ',' startingAt: index.
    type ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
    index ← stop + 1.
    newString release.
    type = 110
    ifTrue:[ aLine ← self getLine: aString.
            self param at: dirNumber put: aLine.]
    ifFalse:[aSurface ← self getSurface: aString.
            aSurface name:((dict at: dirNumber) eLabel).
            self surfaces add: aSurface.]!
```

```
getSurface: aString
    "process a parameter command for a surface"

    | e1 e2 stop index aSurface type newString anEdge anEdge2 |

    index ← 1.
    anEdge ← Line3D new.
    anEdge2 ← Line3D new.
    stop ← aString indexOfSubCollection: ',' startingAt: index.
    type ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
    index ← stop + 1.
    newString release.
    type = 118
    ifTrue:[
        stop ← aString indexOfSubCollection: ',' startingAt: index.
        e1 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        index ← stop + 1.
        newString release.
        stop ← aString indexOfSubCollection: ';' startingAt: index.
        e2 ← (newString ← (aString copyFrom: index to: (stop-1)) asNumber).
        anEdge beginPoint: ((param at: e1) beginPoint deepCopy).
        anEdge endPoint: ((param at: e1) endPoint deepCopy).
        anEdge2 beginPoint: ((param at: e2) beginPoint deepCopy).
        anEdge2 endPoint: ((param at: e2) endPoint deepCopy).
        aSurface ← Surface e1: anEdge e2: anEdge2.
        ↑ aSurface ]
    ifFalse:[ Transcript cr; show: 'Not a Surface'; cr;
                show: 'Entity type = '; print: type;cr; endEntry.]! !
"-- -- -- -
```

```
Object subclass: #Directory
    instanceVariableNames: 'eLabel eType paramDataPntr paramRecCnt dirNumber refByOther '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Directory comment:
'This class is used to hold the directory entry
information from IGES files.'!


!Directory methodsFor: 'accessing'!

dirNumber
    "return the first directory number for the graphic entity"

    ↑dirNumber!

dirNumber: aNumber
    "save the first directory number for the graphic entity"

    dirNumber ← aNumber!

eLabel
    "return the name of the graphic entity"

    ↑eLabel!

eLabel: aString
    "save the name of the graphic entity"

    eLabel ← aString!

eType
    "return the numeric type of the graphic entity"

    ↑eType!

eType: aNumber
    "save the numeric type of the graphic entity"

    eType ← aNumber!

paramDataPntr
    "return the number of the first record of the graphic entity
    in the paramater section"

    ↑paramDataPntr!

paramDataPntr: aNumber
    "save the number of the first record of the graphic entity
    in the paramater section"

    paramDataPntr ← aNumber!

paramRecCnt
```

```
    "return the number of records for the graphic entity
    in the paramater section"

    ↑paramRecCnt!

paramRecCnt:aNumber
    "save the number of records for the graphic entity
    in the paramater section"

    paramRecCnt ← aNumber!

refByOther
    "return if the graphic entity referenced by other entities"

    ↑refByOther!

refByOther:aNumber
    "save status if the graphic entity referenced by other entities"

    refByOther ← aNumber! !
```

```
Object subclass: #Surface
    instanceVariableNames: 'name e1 e2 e3 e4 matrix normal edgenorm ce sina cosa sinb cosb cirot r ri '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Surface comment:
'Each surface is a coplanar ruled surface with straight lines
as its edges. Each edge is a member of the Line3D class. Each
end point on an edge is a vertex. Each vertex is a member of the
Point3D class.'!


!Surface methodsFor: 'accessing'!

ce
    "return common edge ce"

    ↑ ce!

ce: aString
    "save common edge ce"

    ce ← aString!

cirot
    "return the circuit rotation matrix"

    ↑ cirot!

cirot: aMatrix
    "save the circuit rotation matrix"

    cirot ← aMatrix!

cosa
    "return cos of angle"

    ↑ cosa!

cosa: aNumber
    "save cos of angle"

    cosa ← aNumber!

cosb
    "return cos of angle"

    ↑ cosb!

cosb: aNumber
    "save cos of angle"

    cosb ← aNumber!

e1
```

```
    "return edge 1"

    ↑ e1!

e1: aLine3D
    "store edge 1"

    e1 ← aLine3D!

e2
    "return edge 2"

    ↑ e2!

e2: aLine3D
    "store edge 2"

    e2 ← aLine3D!

e3
    "return edge 3"

    ↑ e3!

e3: aLine3D
    "store edge 3"

    e3 ← aLine3D!

e4
    "return edge 4"

    ↑ e4!

e4: aLine3D
    "store edge 4"

    e4 ← aLine3D!

edgenorm
    "return the array of inward edge normals"

    ↑ edgenorm!

edgenorm: anArray
    "save the array of inward edge normals"

    edgenorm ← anArray!

matrix
    "return transformation matrix"

    ↑ matrix!

matrix: aMatrix
    "store transformation matrix"
```

```
    matrix ← aMatrix!

name
    "return name"

    ↑ name!

name: aString
    "save name of surface"

    name ← aString!

normal
    "return normal vector of surface"

    ↑ normal!

normal: aVector
    "save surface normal vector"

    normal ← aVector!

r
    "return the surface rotation matrix"

    ↑ r!

r: aMatrix
    "save the surface rotation matrix"

    r ← aMatrix!

ri
    "return the surface inverse rotation matrix"

    ↑ ri!

ri: aMatrix
    "save the surface inverse rotation matrix"

    ri ← aMatrix!

sina
    "return sin of angle"

    ↑ sina!

sina: aNumber
    "save sin of angle"

    sina ← aNumber!

sinb
    "return sin of angle"
```

```
    ↑ sinb!

sinb: aNumber
    "save sin of angle"

    sinb ← aNumber! !

!Surface methodsFor: 'testing'!

commonEdge: aSurface
    "fine the common edge with another surface"

    (self e1 sameAs: aSurface e1) ifTrue:[ self ce: 'e1'.].
    (self e1 sameAs: aSurface e2) ifTrue:[ self ce: 'e1'.].
    (self e1 sameAs: aSurface e3) ifTrue:[ self ce: 'e1'.].
    (self e1 sameAs: aSurface e4) ifTrue:[ self ce: 'e1'.].
    (self e2 sameAs: aSurface e1) ifTrue:[ self ce: 'e2'.].
    (self e2 sameAs: aSurface e2) ifTrue:[ self ce: 'e2'.].
    (self e2 sameAs: aSurface e3) ifTrue:[ self ce: 'e2'.].
    (self e2 sameAs: aSurface e4) ifTrue:[ self ce: 'e2'.].
    (self e3 sameAs: aSurface e1) ifTrue:[ self ce: 'e3'.].
    (self e3 sameAs: aSurface e2) ifTrue:[ self ce: 'e3'.].
    (self e3 sameAs: aSurface e3) ifTrue:[ self ce: 'e3'.].
    (self e3 sameAs: aSurface e4) ifTrue:[ self ce: 'e3'.].
    (self e4 sameAs: aSurface e1) ifTrue:[ self ce: 'e4'.].
    (self e4 sameAs: aSurface e2) ifTrue:[ self ce: 'e4'.].
    (self e4 sameAs: aSurface e3) ifTrue:[ self ce: 'e4'.].
    (self e4 sameAs: aSurface e4) ifTrue:[ self ce: 'e4'.]! !

!Surface methodsFor: 'private'!

clipPad: aCollectionOfPads
    "Clip all pads into two lists.  Those that fit on the surface
    and those parts which don't.  Cyrus-Beck 2D clipping algorithm."

    | internalCollection externalCollection cnt w wnp col aPad |
    internalCollection ← OrderedCollection new: 1.
    externalCollection ← OrderedCollection new: 1.
    w ← Array new: 4.
    col ← Array new: 2.

    1 to: (aCollectionOfPads size) do:[ :index |
        aPad ← aCollectionOfPads at: index.
        cnt ← 0.
        w at: 1 put: (aPad coordinates - self e1 beginPoint) asVector.
        w at: 2 put: (aPad coordinates - self e2 endPoint) asVector.
        w at: 3 put: (aPad coordinates - self e1 beginPoint) asVector.
        w at: 4 put: (aPad coordinates - self e2 endPoint) asVector.
        1 to: 4 do: [:edg |
            wnp ← (w at: edg) dotProduct: (self edgenorm at: edg).
     "Transcript cr;show:'Pad ';print: index;show:' w ';print:(w at: edg);show:'  wnp ';print: wnp;cr;endEntry."
            (wnp < 0)
            ifTrue:[cnt ← cnt + 1.].
            ].
        (cnt = 0) ifTrue:[ aPad surface: self name.
                           internalCollection add: aPad.]
```

```
                    ifFalse:[externalCollection add: aPad.].
    ].
    col at: 1 put: internalCollection.
    col at: 2 put: externalCollection.
    ↑ col!

clipTrace: aCollectionOfTraces
    "Clip all traces into two lists. Those that fit on the surface
    and those parts which don't. Cyrus-Beck 2D clipping algorithm."

    | internalCollection externalCollection tl tu i k d w
    aTrace aTrace1 aTrace2 aTrace3 dnp wnp t p1 p2 col |
    internalCollection ← OrderedCollection new: 1.
    externalCollection ← OrderedCollection new: 1.
    w ← Array new: 4.
    col ← Array new: 2.
    aTrace1 ← Trace new.
    aTrace2 ← Trace new.
    aTrace3 ← Trace new.
    1 to: (aCollectionOfTraces size) do:[ :index |
        aTrace ← aCollectionOfTraces at: index.
        tl ← 0.
        tu ← 1.
        k ← 0.
        d ← aTrace asVector.
        w at: 1 put: (aTrace beginPoint - self e1 beginPoint) asVector.
        w at: 2 put: (aTrace beginPoint - self e2 beginPoint) asVector.
        w at: 3 put: (aTrace beginPoint - self e3 beginPoint) asVector.
        w at: 4 put: (aTrace beginPoint - self e4 beginPoint) asVector.
    "self halt."
        1 to: 4 do: [:edg |
            dnp ← d dotProduct: (self edgenorm at: edg).
            wnp ← (w at: edg) dotProduct: (self edgenorm at: edg).
            (dnp = 0.0) ifFalse:[
            t ← -1.0*(wnp/dnp).
    "Transcript cr;show:'dnp ';print:dnp;show:' wnp ';print:wnp;show:' t ';print: t;cr;endEntry."
    "Transcript cr;show:'d ';print:d;show:' w ';print:(w at: edg);show:' enorm ';print: (self
        edgenorm at: edg);cr;endEntry."
            (dnp > 0)
            ifTrue:[ (t > 1)
                    ifFalse:[ tl← tl max: t.]
                    ifTrue:[k←1.].]
            ifFalse:[ (t < 0)
                    ifFalse:[ tu ← tu min: t.]
                    ifTrue:[k←1.].].
            ]
            ifTrue:[ (wnp < 0) ifTrue:[k ← 1.].].].
    "Transcript cr; show: 'index ';print: index ; cr;
            show:'tl ';print: tl; cr;
            show: 'tu ';print: tu;cr;endEntry."
    ( k = 1) ifFalse:[
        ( tl > tu)
        ifFalse:[((tl=0)&(tu=1)) "if true whole trace is on surface"
                ifTrue:[ aTrace surface: self name.
                        " set trace as surface dependent?"
                        internalCollection add: aTrace deepCopy.]
```

```
                ifFalse:[
                    p1 ← ((aTrace beginPoint)+(((aTrace endPoint)-(aTrace beginPoint))*tl)).
                    p2 ← ((aTrace beginPoint)+(((aTrace endPoint)-(aTrace beginPoint))*tu)).
                    ((tl > 0) & (tu < 1)) " if true split line into 3 segments"
                  ifTrue:[aTrace1 beginPoint: aTrace beginPoint.
                          aTrace1 endPoint: p1.
                          aTrace1 w: aTrace w.
                          aTrace2 beginPoint: p1.
                          aTrace2 endPoint: p2.
                          aTrace2 w: aTrace w.
                          aTrace2 surface: self name.
                          aTrace3 beginPoint: p2.
                          aTrace3 endPoint: aTrace endPoint.
                          aTrace3 w: aTrace w.
                          internalCollection add: aTrace2 deepCopy.
                          externalCollection add: aTrace1 deepCopy.
                          externalCollection add: aTrace3 deepCopy.]
                   ifFalse:[((tl=0)&(tu < 1))
                          ifTrue:[aTrace2 beginPoint: aTrace beginPoint.
                              aTrace2 endPoint: p2.
                              aTrace2 w: aTrace w.
                              aTrace2 surface: self name.
                              aTrace3 beginPoint: p2.
                              aTrace3 endPoint: aTrace endPoint.
                              aTrace3 w: aTrace w.
                              internalCollection add: aTrace2 deepCopy.
                              externalCollection add: aTrace3 deepCopy.]
                          ifFalse:[aTrace2 beginPoint: p1.
                          aTrace2 endPoint: aTrace endPoint.
                          aTrace2 w: aTrace w.
                          aTrace2 surface: self name.
                          aTrace3 beginPoint: aTrace beginPoint.
                          aTrace3 endPoint: p1.
                          aTrace3 w: aTrace w.
                          internalCollection add: aTrace2 deepCopy.
                          externalCollection add: aTrace3 deepCopy.].].
                          ].].
            ]
        ifTrue:[externalCollection add: aTrace deepCopy.].
].
    col at: 1 put: internalCollection.
    col at: 2 put: externalCollection.
    ↑ col!

findAngle: aSurface
    "find angles for rotation between surfaces"

    | v1 v2 v3 v4 x y|
    x ← Vector x: 1.0 y: 0.0 z: 0.0 w: 0.0.
    y ← Vector x: 0.0 y: 1.0 z: 0.0 w: 0.0.
    v1 ← self normal cos: aSurface normal.
    v2 ← self normal sin: aSurface normal.
    v3 ← x cos: aSurface normal.
    v4 ← x sin: aSurface normal.
    "Transcript cr;show:'sina ';print: v2;show:' cosa ';print:v1;cr;endEntry.
    Transcript cr;show:'sinb ';print: v4;show:' cosb ';print:v3;cr;endEntry."
```

```
    self cosa: v1.
    self sina: v2.
    self cosb: v3.
    self sinb: v4!

findNormal
    "calculate the inward normal vector of an edge"

    | v1 v2 v3 v4 p1 p2 p3 p4 q1 q2 q3 q4 anArray|

    anArray ← Array new: 4.
    v1 ← self e1 asVector.
    v2 ← self e2 asVector.
    v3 ← self e3 asVector.
    v4 ← self e4 asVector.
    p1 ← (Point3D x:(-1.0*v1 y) y: (v1 x) z:(v1 z)) asVector.
    p2 ← (Point3D x:(-1.0*v2 y) y: (v2 x) z:(v2 z)) asVector.
    p3 ← (Point3D x:(-1.0*v3 y) y: (v3 x) z:(v3 z)) asVector.
    p4 ← (Point3D x:(-1.0*v4 y) y: (v4 x) z:(v4 z)) asVector.
    q1 ← ((self  e2 endPoint) - (self  e1 beginPoint)) asVector.
    q2 ← ((self  e1 endPoint) - (self  e2 beginPoint)) asVector.
    q3 ← ((self  e2 endPoint) - (self  e3 beginPoint)) asVector.
    q4 ← ((self  e2 beginPoint) - (self  e4 beginPoint)) asVector.
    ((p1 dotProduct: q1) < 0)
    ifTrue:[p1 x: (-1.0 *(p1 x)).
            p1 y: (-1.0*(p1 y)).].
    ((p2 dotProduct: q2) < 0)
    ifTrue:[p2 x: (-1.0 *(p2 x)).
            p2 y: (-1.0*(p2 y)).].
    ((p3 dotProduct: q3) < 0)
    ifTrue:[p3 x: (-1.0 *(p3 x)).
            p3 y: (-1.0*(p3 y)).].
    ((p4 dotProduct: q4) < 0)
    ifTrue:[p4 x: (-1.0 *(p4 x)).
            p4 y: (-1.0*(p4 y)).].
    anArray at: 1 put: (p1 unit).
    anArray at: 2 put: (p2 unit).
    anArray at: 3 put: (p3 unit).
    anArray at: 4 put: (p4 unit).
    ↑ anArray!

form: aForm
    "set the form for line width for the 2D edge lines"

    self e1:(self e1 form: aForm).
    self e2:(self e2 form: aForm).
    self e3:(self e3 form: aForm).
    self e4:(self e4 form: aForm).
    ↑ self! !

!Surface methodsFor: 'transforming'!

buildRotX: costh with: sinth
    "build a rotation matrix for the X axis."

    | m n a|
```

```
    a ← Array new: 2.
    m ← Matrix new:4 by: 4.
    m setToZero.
    m atPoint: 1@1 put: 1.0.
    m atPoint: 2@2 put: costh.
    m atPoint: 3@2 put: sinth.
    m atPoint: 2@3 put:(-1.0 * sinth).
    m atPoint: 3@3 put: costh.
    m atPoint: 4@4 put: 1.0.
    n ← m fromPoint: 1@1 toPoint: 4@4.
    n atPoint: 3@2 put:(-1.0 * sinth).
    n atPoint: 2@3 put: sinth.
    a at: 1 put: m.
    a at: 2 put: n.
    ↑ a!

buildRotY: costh with: sinth
    "build a rotation matrix for the Y axis."

    | m n a |

    a ← Array new: 2.
    m ← Matrix new:4 by: 4.
    m setToZero.
    m atPoint: 1@1 put: costh.
    m atPoint: 3@1 put: (-1.0 * sinth).
    m atPoint: 2@2 put: 1.0.
    m atPoint: 1@3 put:sinth.
    m atPoint: 3@3 put: costh.
    m atPoint: 4@4 put: 1.0.
    n ← m fromPoint: 1@1 toPoint: 4@4.
    n atPoint: 3@1 put: sinth.
    n atPoint: 1@3 put:(-1.0 * sinth).
    a at: 1 put: m.
    a at: 2 put: n.
    ↑ a!

buildRotZ: costh with: sinth
    "build a rotation matrix for the Z axis."

    | m n a |

    a ← Array new: 2.
    m ← Matrix new:4 by: 4.
    m setToZero.
    m atPoint: 1@1 put: costh.
    m atPoint: 2@1 put: sinth.
    m atPoint: 1@2 put: (-1.0 * sinth).
    m atPoint: 2@2 put:costh.
    m atPoint: 3@3 put: 1.0.
    m atPoint: 4@4 put: 1.0.
    n ← m fromPoint: 1@1 toPoint: 4@4.
    n atPoint: 2@1 put: (-1.0 * sinth).
    n atPoint: 1@2 put: sinth.
    a at: 1 put: m.
```

```
    a at: 2 put: n.
    ↑ a!

buildRotZc: aMatrix
    "build a special circuit rotation matrix for the Z axis."

    | m s c n a |

    a ← Array new: 2.
    m ← aMatrix fromPoint: 1@1 toPoint: 4@4.
    s ← m atPoint: 2@1 deepCopy.
    c ← m atPoint: 1@1 deepCopy.
    m atPoint: 1@1 put: s.
    m atPoint: 2@2 put: s.
    m atPoint: 2@1 put: c.
    m atPoint: 1@2 put: (-1.0 * c).
    n ← m fromPoint: 1@1 toPoint: 4@4.
    n atPoint: 2@1 put: ((n atPoint: 2@1)*-1.0).
    n atPoint: 1@2 put: ((n atPoint: 1@2)*-1.0).
    a at: 1 put: m.
    a at: 2 put: n.
    ↑ a!

buildTrans: aPoint3D
    "build a translation matrix from a 3D point."

    | t m a|

    a ← Array new: 2.
    t ← Matrix new:4 by: 4.
    t setToZero.
    t atPoint: 1@4 put: (( aPoint3D x) * -1.0).
    t atPoint: 2@4 put: (( aPoint3D y) * -1.0).
    t atPoint: 3@4 put: (( aPoint3D z) * -1.0).
    t atPoint: 4@4 put: 1.0.
    t atPoint: 1@1 put: 1.0.
    t atPoint: 2@2 put: 1.0.
    t atPoint: 3@3 put: 1.0.
    m ← t fromPoint: 1@1 toPoint: 4@4.
    m atPoint: 1@4 put:(aPoint3D x).
    m atPoint: 2@4 put:(aPoint3D y).
    m atPoint: 3@4 put:(aPoint3D z).
    a at: 1 put: t.
    a at: 2 put: m.
    ↑ a!

calcRotation: aSurface
    "calculate circuit and surface rotations for mapping to aSurface"

    | t ti tc tci rx rxi ry ryi rz rzi rzc rzci rztmp tmp |
    tmp ← self buildTrans:(aSurface e1 beginPoint).
    t ← tmp at: 1.
    ti ← tmp at: 2.
    ((self ce = 'e1') | (self ce ='e2'))
    ifTrue:[
        (self ce = 'e1') ifTrue:[
```

```
            tmp ← self buildTrans:(self e1 beginPoint).]
        ifFalse:[ tmp ← self buildTrans:(self e2 beginPoint).].
        tc ← tmp at: 1.
        tci ← tmp at: 2.
        tmp ← self buildRotX: self cosa with: self sina.
        rx ← tmp at: 1.
        rxi ← tmp at: 2.
        "self halt."
        (self cosb < 0) ifTrue:[
            self sinb: self sinb * -1.0.
            self cosb: self cosb * -1.0.].
        ((self sinb = 1) | (self sinb = -1)) ifTrue:[ self cosb: 1.0.
                                    self sinb: 0.0.].
        tmp ← self buildRotZ: self cosb with: self sinb.
        rz ← tmp at: 1.
        rzi ← tmp at: 2.
        ( (aSurface normal y) >= 0) ifTrue:[
            self cirot: tc * rz * rxi * tci.   " the order is crucial !!!!!!"
            self r:  t * rz * rx.
            self ri: rxi * rzi * ti.]
        ifFalse:[
            self cirot: tc * rzi * rx * tci.
            self r: t * rzi * rxi.
            self ri: rx * rz * ti.].
    "self halt."]
    ifFalse:[
        (self ce = 'e3') ifTrue:[
            tmp ← self buildTrans:(self e3 beginPoint).]
        ifFalse:[ tmp ← self buildTrans:(self e4 beginPoint).].
        tc ← tmp at: 1.
        tci ← tmp at: 2.
        tmp ← self buildRotY: self cosa with: self sina.
        ry ← tmp at: 1.
        ryi ← tmp at: 2.
        (self sinb < 0) ifTrue:[
            self sinb: self sinb * -1.0.
            self cosb: self cosb * -1.0.].
        (self cosb = -1) ifTrue:[ self cosb: self cosb * -1.0.].
        tmp ← self buildRotZ: self cosb with: self sinb.
        rz ← tmp at: 1.
        rzi ← tmp at: 2.
        ( (aSurface normal x) >= 0) ifTrue:[
            self cirot: tc * rzi * ry * tci.   " the order is crucial !!!!!!"
            self r:  t * rzi * ryi.
            self ri: ry * rz * ti.]
        ifFalse:[
            self cirot: tc * rzi * ryi * tci.
            self r: t * rz * ry.
            self ri: ryi * rzi * ti.].
        "self halt."].!

convToView: a3DPoint
    "convert world coordinates to view coordinates based on world extents
    a3DPoint is the extents of the world view."

    | aSurface point1 point2 nTrace wPoint aLine aLine2|
```

```
    aLine ← Line3D new.
    aLine2 ← Line3D new.
    wPoint ← 1024.0@800.0@800.0.
    point1 ← self e1 beginPoint. "begin translation from world to display coordinates"
    point2 ← self  e1 endPoint.
    point1 ← (point1 / a3DPoint) * wPoint.  " constant is view maximums"
    point2 ← (point2 / a3DPoint) * wPoint. "assume z extent in view same as y"
    point1 y: ( 0.0 - (point1 y)). "invert y to move display origin to lower left"
    point2 y: (0.0 - (point2 y)).
    aLine beginPoint: point1.
    aLine endPoint: point2.
    point1 ← self e2 beginPoint. "begin translation from world to display coordinates"
    point2 ← self  e2 endPoint.
    point1 ← (point1 / a3DPoint) * wPoint.  " constant is view maximums"
    point2 ← (point2 / a3DPoint) * wPoint. "assume z extent in view same as y"
    point1 y: ( 0.0 - (point1 y)). "invert y to move display origin to lower left"
    point2 y: (0.0 - (point2 y)).
    aLine2 beginPoint: point1.
    aLine2 endPoint: point2.
    aSurface ← Surface e1: aLine e2: aLine2.
    ↑ aSurface!

mapToNextSurface: traceList with: padList and: surfaces and: surflist
    "This is the heart of mapping program.  Clip all circuitry
    for current circuitry.  The according to surflist, map remaining circuitry
    to attached surfaces.  Include their mapped circuitry in current list.
    Return mapped circuitry to calling object."

    | mappedCircuits internalTraces externalTraces internalPads
    externalPads tempCircuit tempTrace tempPad linkList nextSurface sname |
    mappedCircuits ← Array new:2.
    Transcript cr; show: 'Mapping surface: ';print: self name;cr;endEntry.
    self edgenorm: ( self findNormal).
    tempCircuit ← self clipTrace: traceList.
    internalTraces ← (tempCircuit at: 1)deepCopy.
    externalTraces ← (tempCircuit at: 2)deepCopy.
    tempCircuit ← self clipPad: padList.
    internalPads ← (tempCircuit at: 1)deepCopy.
    externalPads ← (tempCircuit at: 2)deepCopy.
    "self halt."
    1 to:(surflist size) do:[:index |
    ((surflist at: index) = nil) ifFalse:[
    (self name = ((surflist at: index) first name)) ifTrue:[
        linkList ← surflist at: index.
        2 to: linkList size do:[:cnt |
            sname ← (linkList at: cnt) name.
        1 to: (surfaces size) do:[:scnt |
        ((surfaces at: scnt) name = sname) ifTrue:[
            nextSurface ← surfaces at: scnt.].].
            self findAngle: nextSurface.
            self commonEdge: nextSurface.
            self calcRotation: nextSurface.
            tempTrace ← externalTraces deepCopy.
            1 to: (tempTrace size) do:[:tl
                (tempTrace at: t) rotate: self cirot.].
```

```
            tempPad ← externalPads deepCopy.
            1 to: (tempPad size) do:[:t|
                (tempPad at: t) rotate: self cirot.].
            1 to: (surfaces size) do:[:t|
            (surfaces at: t) rotate: self r.].
            1 to: (tempTrace size) do:[:t|
                (tempTrace at: t) rotate: self r.].
            1 to: (tempPad size) do:[:t|
                (tempPad at: t) rotate: self r.].
            tempCircuit ← nextSurface mapToNextSurface: tempTrace
                                with: tempPad and: surfaces and: surflist.
            1 to: (surfaces size) do:[:t|
            (surfaces at: t) rotate: self ri.].
            tempTrace ← tempCircuit at: 1.
            tempPad ← tempCircuit at: 2.
            1 to: (tempTrace size) do:[:t|
                (tempTrace at: t) rotate: self ri.].
            1 to: (tempPad size) do:[:t|
                (tempPad at: t) rotate: self ri.].
            internalTraces addAllLast: tempTrace deepCopy.
            internalPads addAllLast: tempPad deepCopy.
        ].
    ].].].
    mappedCircuits at: 1 put: internalTraces deepCopy.
    mappedCircuits at: 2 put: internalPads deepCopy.
    ↑ mappedCircuits!

rotate: aMatrix
    "rotate a surface as defined by a transformation matrix"

    | m1 m2 m3 m4 m5 |
    m1 ← self e1 beginPoint asMatrix.
    m2 ← self e1 endPoint asMatrix.
    m3 ← self e2 beginPoint asMatrix.
    m4 ← self e2 endPoint asMatrix.
    m5 ← self normal asMatrix.
    m1 ← m1*aMatrix.
    m2 ← m2*aMatrix.
    m3← m3*aMatrix.
    m4← m4*aMatrix.
    m5 ← m5 * aMatrix.
    self e1 beginPoint: m1 asPoint3D.
    self e3 beginPoint: m1 asPoint3D.
    self e1 endPoint: m2 asPoint3D.
    self e4 beginPoint: m2 asPoint3D.
    self e2 beginPoint: m3 asPoint3D.
    self e3 endPoint: m3 asPoint3D.
    self e2 endPoint: m4 asPoint3D.
    self e4 endPoint: m4 asPoint3D.
    self normal: m5 asVector!

scaleBy: aPoint
    "scale 2D edges of a surface for viewing"

    self e1: (self e1 scaleBy: aPoint).
    self e2: (self e2 scaleBy: aPoint).
```

```
    self e3: (self e3 scaleBy: aPoint).
    self e4: (self e4 scaleBy: aPoint).
    ↑ self! !

!Surface methodsFor: 'displaying'!

displayOn: aDisplayMedium at: aPoint clippingBox: clipRect
    "The form associated with this Surface will be displayed, according
    to one of the sixteen functions of two logical variables (rule), at
    each point on the Line. Also the source form will be first ANDed
    with aForm as a mask. Does not effect the state of the Path."


    self e1 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect.
    self e2 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect.
    self e3 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect.
    self e4 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect!

displayOn: aDisplayMedium at: aPoint clippingBox: clipRect rule: anInteger mask: aForm
    "The form associated with this Surface will be displayed, according
    to one of the sixteen functions of two logical variables (rule), at
    each point on the Line. Also the source form will be first ANDed
    with aForm as a mask. Does not effect the state of the Path."


    self e1 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect rule: anInteger mask: aForm.
    self e2 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect rule: anInteger mask: aForm.
    self e3 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect rule: anInteger mask: aForm.
    self e4 displayOn: aDisplayMedium at: aPoint clippingBox: clipRect rule: anInteger mask: aForm

"   aDisplayMedium
        drawLine: e1 form
        from: e1 beginPoint + aPoint
        to: e1 endPoint + aPoint
        clippingBox: clipRect
        rule: anInteger
        mask: aForm.
    aDisplayMedium
        drawLine: e2 form
        from: e2 beginPoint + aPoint
        to: e2 endPoint + aPoint
        clippingBox: clipRect
        rule: anInteger
        mask: aForm.
    aDisplayMedium
        drawLine: e3 form
        from: e3 beginPoint + aPoint
        to: e3 endPoint + aPoint
        clippingBox: clipRect
        rule: anInteger
        mask: aForm.
    aDisplayMedium
        drawLine: e4 form
        from: e4 beginPoint + aPoint
        to: e4 endPoint + aPoint
        clippingBox: clipRect
```

```
        rule: anInteger
        mask: aForm"! !
"-- -- -- -
```

```
Link subclass: #SurfList
    instanceVariableNames: 'name '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
SurfList comment:
'This class is the elements of the surface list
which orders the mapping process.'!


!SurfList methodsFor: 'printing'!

printOn: aStream
    "print surface name"

    aStream nextPutAll: 'Surface name: ', name printString! !

!SurfList methodsFor: 'accessing'!

name
    "retrieve the name of a surface"

    ↑ name!

name: aString
    "save the name of a surface"

    ↑ name ← aString! !
```

```
'From Smalltalk-80, Version 2.3 of 13 June 1988 on 21 August 1989 at 9:03:04 pm'!

Point subclass: #Point3D
    instanceVariableNames: 'z '
    classVariableNames: 'RandomSource '
    poolDictionaries: ''
    category: 'Graphics-Primitives'!


!Point3D methodsFor: 'accessing'!

z
    ↑ z!

z: value
    ↑ z ← value! !

!Point3D methodsFor: 'arithmetic'!

* scale
    "Answer a new Point that is the product of the receiver and scale (which is a Point
    or Number)."

    | scalePoint |
    (scale isMemberOf: Point3D)
    ifFalse:[ scalePoint ← scale@scale@scale.]
    ifTrue:[  scalePoint ← scale as3DPoint.].
    ↑x * scalePoint x @ (y * scalePoint y) @ ( z * scalePoint z)!

+ delta
    "Answer a new Point that is the sum of the receiver and delta (which is a Point
    or Number)."

    | deltaPoint |
    deltaPoint ← delta as3DPoint.
    ↑x + deltaPoint x @ (y + deltaPoint y) @ ( z + deltaPoint z)!

- delta
    "Answer a new Point that is the difference of the receiver and delta (which is a Point
    or Number)."

    | deltaPoint |
    deltaPoint ← delta as3DPoint.
    ↑x - deltaPoint x @ (y - deltaPoint y) @ ( z - deltaPoint z)!

/ scale
    "Answer a new Point that is the quotient of the receiver and scale (which is a Point
    or Number)."

    | scalePoint |
    scalePoint ← scale as3DPoint.
    ↑x / scalePoint x @ (y / scalePoint y) @ ( z / scalePoint z)!

// scale
    "Answer a new Point that is the quotient of the receiver and scale (which is a Point
    or Number)."
```

```
    | scalePoint |
    scalePoint ← scale as3DPoint.
    ↑x // scalePoint x @ (y // scalePoint y) @ ( z // scalePoint z)!

abs
    "Answer a new Point whose x and y are the absolute values of the receiver's
    x and y."

    ↑Point3D x: x abs y: y abs z: z abs! !

!Point3D methodsFor: 'converting'!

as2DPoint

    "Answer with 2D version of self"

    | newPoint nx ny|
    nx ← self x.
    ny ← self y.
    newPoint ← x@y.
    ↑newPoint!

as3DPoint

    "Answer the receiver itself."

    ↑self!

asMatrix

    "Answer with a matrix version of self"

    | mat |
    mat ← Matrix new: 1 by:4.
    mat atPoint: 1@1 put: self x.
    mat atPoint: 1@2 put: self y.
    mat atPoint: 1@3 put: self z.
    mat atPoint: 1@4 put: 1.0.
    ↑ mat!

asVector

    "Answer with a vector version of self"

    | newVector nx ny nz|
    nx ← self x.
    ny ← self y.
    nz ← self z.
    newVector ← Vector x: nx y: ny z: nz w: 1.0.
    ↑newVector!

middle: otherPoint displaced: range
    "Answer a point in the middle, randomly displaced.
    Keep coordinates integral for speed only."
```

```
    | newX newY newZ |
    newX ← (x + otherPoint x) // 2.
    newY ← (y + otherPoint y) // 2.
    newZ ← (z + otherPoint z) // 2.
    newZ ← newZ + (RandomSource next - 0.5 * range) truncated.
    ↑ Point3D x: newX y: newY z: newZ! !

!Point3D methodsFor: 'printing'!

printOn: aStream
    "The receiver prints on aStream in terms of infix notation."

    x printOn: aStream.
    aStream nextPut: $@.
    y printOn: aStream.
    aStream nextPut: $@.
    z printOn: aStream! !

!Point3D methodsFor: 'private'!

setX: xValue setY: yValue setZ: zValue
    x ← xValue.
    y ← yValue.
    z ← zValue! !

!Point3D methodsFor: 'comparing'!

< aPoint
    "Answer whether the receiver is 'above and to the left and behind' of the
    argument, aPoint."

    ↑(x < aPoint x and: [y < aPoint y]) and: [z < aPoint z]!

<= aPoint
    "Answer whether the receiver is 'neither below and to the right and in front' of the
    argument, aPoint."

    ↑(x <= aPoint x and: [y <= aPoint y]) and: [z <= aPoint z]!

= aPoint
    "Answer whether the receiver is 'equal' to the
    argument, aPoint."

    ↑(x = aPoint x and: [y = aPoint y]) and: [z = aPoint z]!

> aPoint
    "Answer whether the receiver is 'below and to the right and in front' of the
    argument, aPoint."

    ↑(x > aPoint x and: [y > aPoint y]) and: [z > aPoint z]!

>= aPoint
    "Answer whether the receiver is 'neither above and to the left and behind' of the
    argument, aPoint."

    ↑(x >= aPoint x and: [y >= aPoint y]) and: [z >= aPoint z]! !
```

```
!Point3D methodsFor: 'vector'!

xprd: aPoint3D
    "Treat two 3D points as vectors and return their cross product
    as a 3D point.  This will be the normal vector for the first two vectors."

    | i j k x1 y1 z1 |! !

!Point3D methodsFor: 'transformation'!

convToView: a3DPoint
    "convert world coordinates to view coordinates based on world extents
    a3DPoint is the extents of the world view."

    | point1 |

    point1 ← (self / a3DPoint) * 1000@800@800.  " constant is view maximums"
    point1 y: (800 - (point1 y)).  "invert y to move display origin to lower left"
    ↑ point1! !

!Point3D methodsFor: 'copying'!

deepCopy
    "Answer a copy of the receiver with its own copy of each instance variable."

    "Implemented here for better performance."

    ↑x deepCopy @ y deepCopy @ z deepCopy!

shallowCopy
    "Implemented here for better performance."

    ↑x @ y @ z! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Point3D class
    instanceVariableNames: ''!


!Point3D class methodsFor: 'class initialization'!

initialize
    RandomSource ← Random new

    "Point3D initialize"! !

!Point3D class methodsFor: 'instance creation'!

x: xValue y: yValue z: zValue
    "Answer an instance of me with coordinates xValue, yValue, and zValue."

    ↑self new setX: xValue setY: yValue setZ: zValue! !

Point3D initialize!
```

```
'From Smalltalk-80, Version 2.3 of 13 June 1988 on 21 August 1989 at 11:55:42 am'!

Line subclass: #Line3D
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Paths'!


!Line3D methodsFor: 'displaying'!

displayOn: aDisplayMedium at: aPoint clippingBox: clipRect rule: anInteger mask: aForm
    "The form associated with this Path will be displayed, according
    to one of the sixteen functions of two logical variables (rule), at
    each point on the Line. Also the source form will be first ANDed
    with aForm as a mask. Does not effect the state of the Path."

    collectionOfPoints size < 2 ifTrue: [self error: 'a line must have two points'].
    aDisplayMedium
        drawLine: self form
        from: (self beginPoint) as2DPoint + aPoint
        to: (self endPoint) as2DPoint + aPoint
        clippingBox: clipRect
        rule: anInteger
        mask: aForm! !

!Line3D methodsFor: 'comparing'!

<= aLine
    "compare the starting points of a 3D line"

    ↑ self beginPoint <= aLine beginPoint! !

!Line3D methodsFor: 'converting'!

as2DLine
    "convert 3D line to 2D line by romoving z coordinates"

    |tx ty aLine|
    aLine ← Line new.
    tx ← self beginPoint x.
    ty ← self beginPoint y.
    aLine beginPoint: tx@ty.
    tx ← self endPoint x.
    ty ← self endPoint y.
    aLine endPoint: tx@ty.
    ↑aLine!

asVector
    "convert 3D line to a vector"

    |x y z aVector|

    x ← (self endPoint x) - (self beginPoint x).
    y ← (self endPoint y) - (self beginPoint y).
    z ← (self endPoint z) - (self beginPoint z).
```

```
    aVector ← Vector x: x y: y z: z w: 1.0.
    ↑ aVector! !

!Line3D methodsFor: 'transforming'! !

!Line3D methodsFor: 'testing'!

sameAs: aLine3D
    "see if lines are the same"

    (self beginPoint = aLine3D beginPoint)
    ifTrue:[(self endPoint = aLine3D endPoint)
        ifTrue:[ ↑ true].].
    ↑ false! !
"-- -- -- -
```

```
Point3D subclass: #Vector
    instanceVariableNames: 'w '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Graphics-Circuits'!
Vector comment:
'This class are vectors for manipulating surfaces.'!


!Vector methodsFor: 'mathematical functions'!

angle: aVector
    "find the angle between two vectors."

    | theta c |
    c ← self cos: aVector.
    ↑ theta ← (c arcCos) / 0.0174532925!

cos: aVector
    "find the cosine of the angle between two vectors."

    | c d m1 m2 |
    d ← self dotProduct: aVector.
    m1 ← self magnitude.
    m2 ← aVector magnitude.
    ↑ c ← d / (m1 * m2)!

projection: aVector
    "find the projection of the second vector onto the first vector."

    | aNewVector d1 d2 |
    d1 ← (self dotProduct: aVector).
    d2 ← (self dotProduct: self).
    ↑ aNewVector ← self * (d1/d2)!

sin: aVector
    "find the sine of the angle between two vectors."

    | c d m1 m2 |
    d ← (self xProduct: aVector) magnitude.
    m1 ← self magnitude.
    m2 ← aVector magnitude.
    ↑ c ← d / (m1 * m2)!

unit
    "find the unit vector of a vector."

    | aVector m |
    m ← self magnitude.
    ↑ aVector ← self / m! !

!Vector methodsFor: 'arithmetic'!

* aScalar
    "Find the vector which is the product of a vector and a scalar."
```

```
    | aVector |
    aVector ← Vector x: (self x * aScalar)
                     y: (self y * aScalar)
                     z: (self z * aScalar)
                     w: (self w * aScalar).
    ↑ aVector!

/ aScalar
    "Find the vector which is the quotient of a vector and a scalar."

    | aVector |
    aVector ← Vector x: (self x / aScalar)
                     y: (self y / aScalar)
                     z: (self z / aScalar)
                     w: (self w / aScalar).
    ↑ aVector!

dotProduct: aVector
    "Find the dot product of two vectors. The result is a scalar."

    | a b c |
    a ← self x * aVector x.
    b ← self y * aVector y.
    c ← self z * aVector z.
    ↑ a ← a +b +c!

magnitude
    "Find the magnitude of a vector. The result is a positive scalar value."

    | m |
    m ← self dotProduct: self.
    ↑ m sqrt!

xProduct: aVector
    "Find the cross product of two vectors. The result is the normal vector
    of the two."

    | i j k a b c d e f |
    a ← self y * aVector z.
    b ← self z * aVector x.
    c ← self x * aVector y.
    a ← a - (self z * aVector y).
    b ← b - (self x * aVector z).
    c ← c - (self y * aVector x).
    ↑ Vector x: a y: b z: c w: 0.0! !

!Vector methodsFor: 'accessing'!

w
    "return w value for homogeneous representation"

    ↑ w!

w: aFloat
    "save w value for homogeneous representation"
```

```
    ↑ w ← aFloat!

x
    "return x value for homogeneous representation"

    ↑ x!

x: aFloat
    "save x value for homogeneous representation"

    ↑ x ← aFloat!

y
    "return y value for homogeneous representation"

    ↑ y!

y: aFloat
    "save y value for homogeneous representation"

    ↑ y ← aFloat!

z
    "return z value for homogeneous representation"

    ↑ z!

z: aFloat
    "save z value for homogeneous representation"

    ↑ z ← aFloat! !

!Vector methodsFor: 'printing'!

printOn: aStream
    "The receiver prints on aStream in terms of infix notation."

    x printOn: aStream.
    aStream nextPut: $@.
    y printOn: aStream.
    aStream nextPut: $@.
    z printOn: aStream.
    aStream nextPut: $@.
    w printOn: aStream! !

!Vector methodsFor: 'converting'!

asMatrix

    "Answer with a matrix version of self"

    | mat |
    mat ← Matrix new: 1 by: 4.
    mat atPoint: 1@1 put: self x.
    mat atPoint: 1@2 put: self y.
```

```
    mat atPoint: 1@3 put: self z.
    mat atPoint: 1@4 put: self w.
    ↑ mat! !
"-- -- -- -
```

```
'From Smalltalk-80, Version 2.3 of 13 June 1988 on 1 September 1989 at 4:01:34 pm'!


!Matrix methodsFor: 'converting'!

asPoint3D
    "convert a matrix to a 3D point"

    | point |
    point ← Point3D new.
    point x: (self atPoint: 1@1).
    point y: (self atPoint: 1@2).
    point z: (self atPoint: 1@3).
    ↑ point!

asVector
    "convert a matrix to a vector"

    | v |
    v ← Vector x:(self atPoint: 1@1)
                y: (self atPoint: 1@2)
                z: (self atPoint: 1@3)
                w: (self atPoint: 1@4).
    ↑ v! !
```

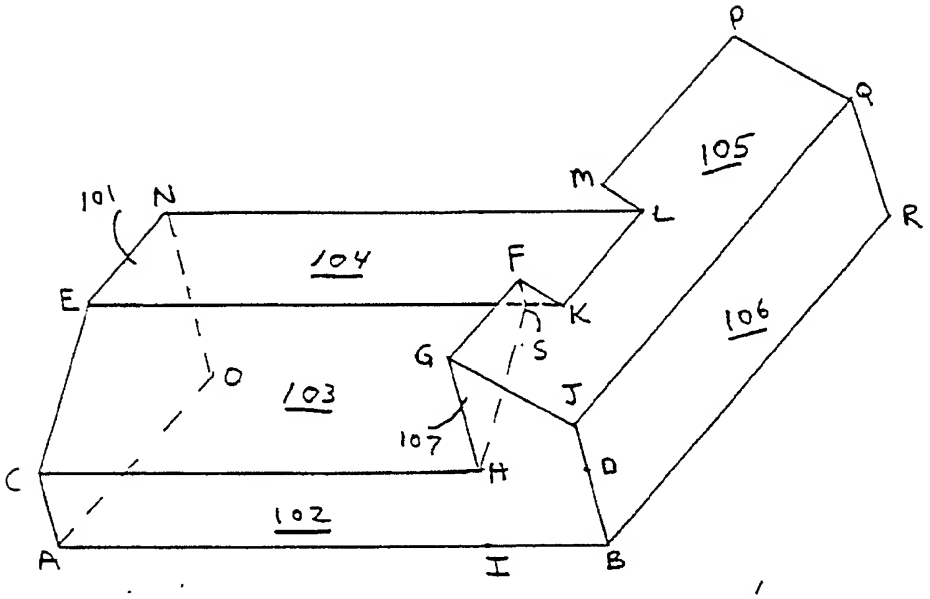
P
Q
R
105
M
L
101
N
104
F
E
K
106
S
G
O
103
J
107
H
D
C
102
A
I
B

FIG. 1

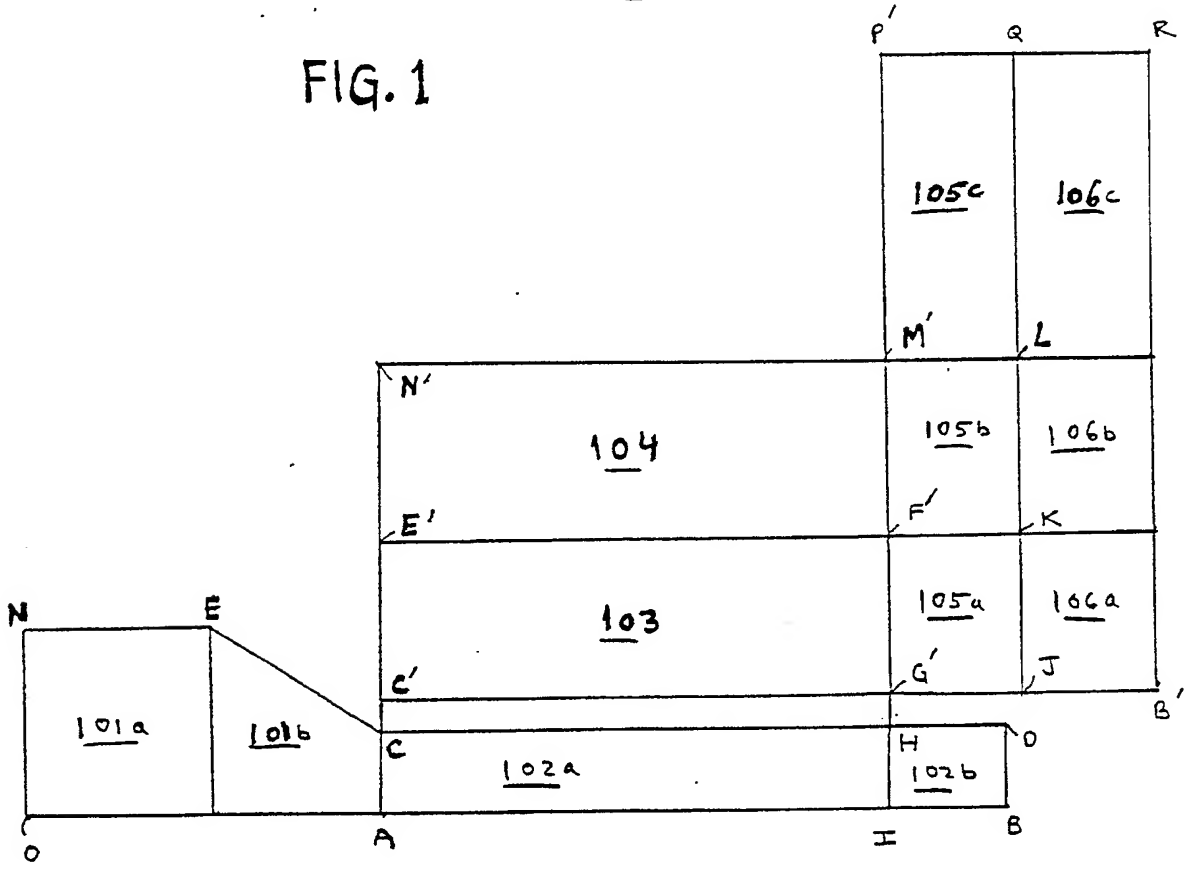
P'
Q
R
105c
106c
M'
L
N'
104
105b
106b
E'
F'
K
103
105a
106a
C'
G'
J
B'
N
E
101a
101b
C
H
O
102a
102b
O
A
I
B

FIG. 2

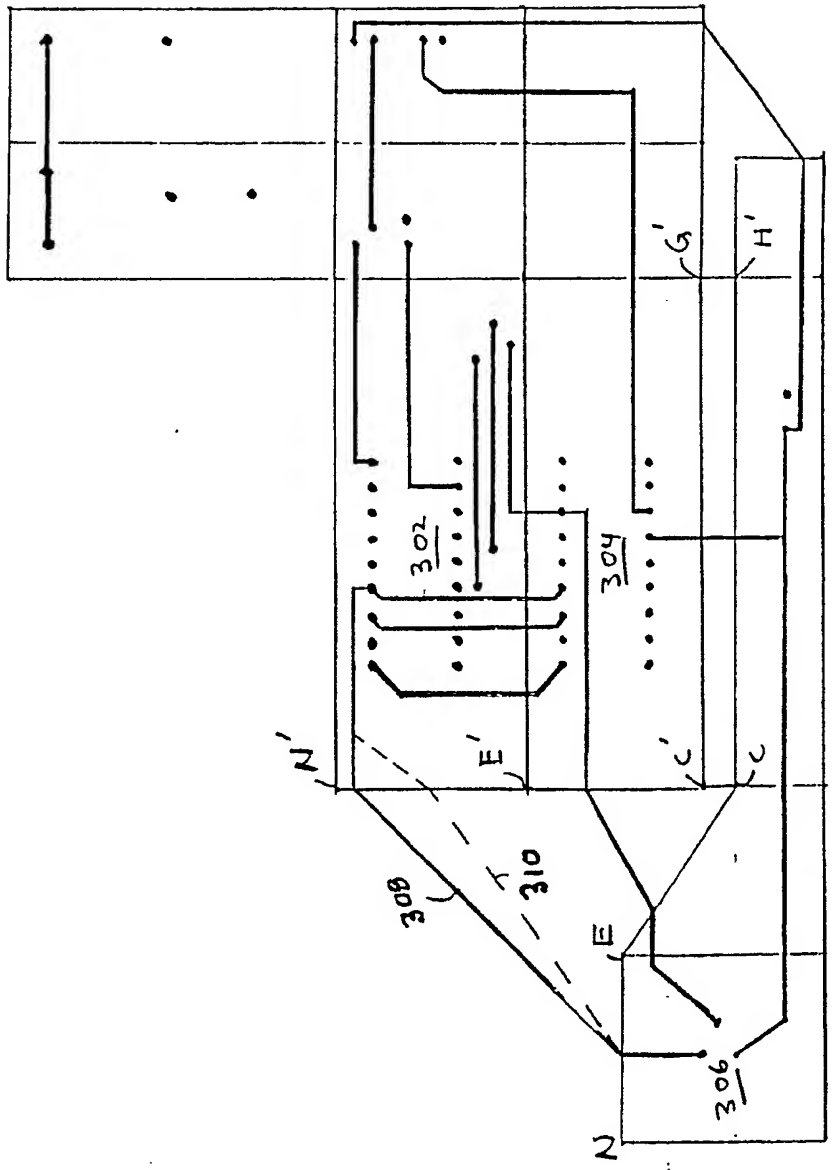

FIG.3

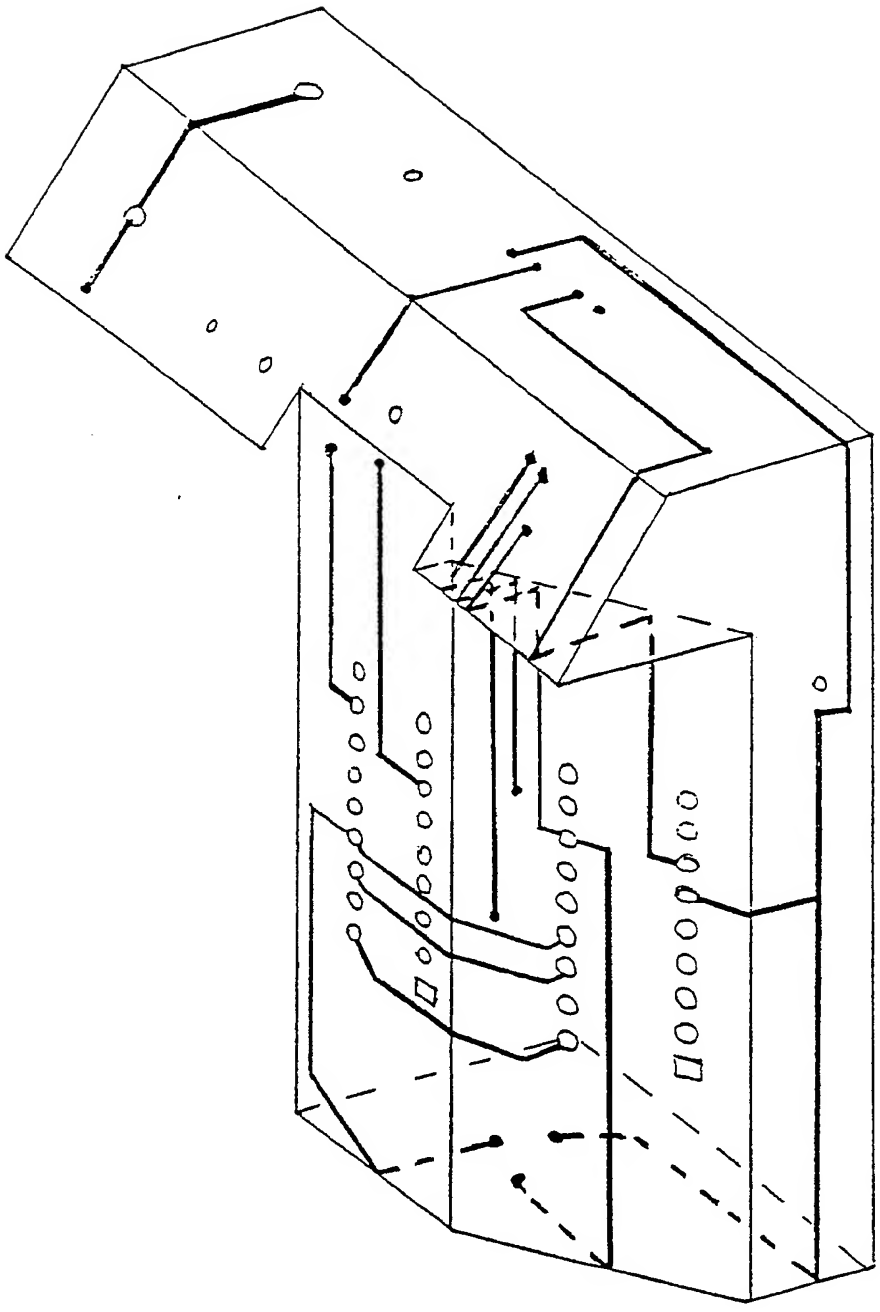

FIG. 4

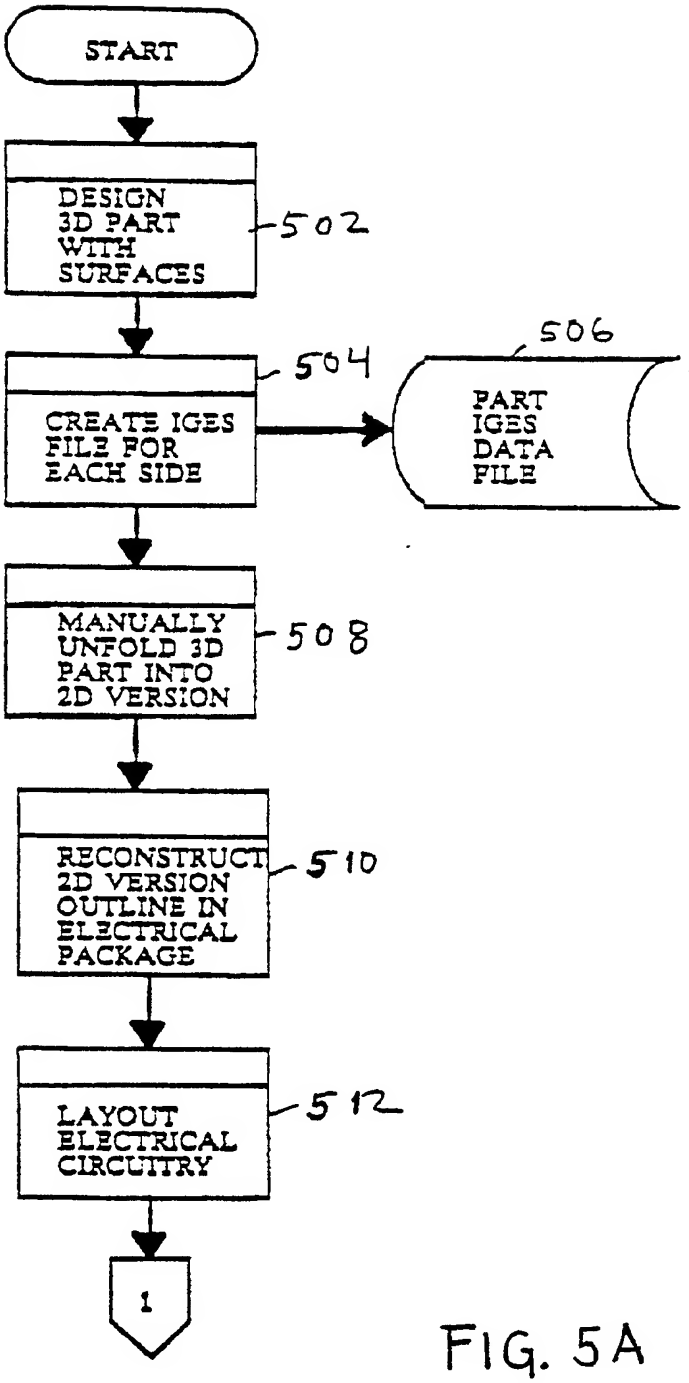
START
DESIGN 3D PART WITH SURFACES
502
504
CREATE IGES FILE FOR EACH SIDE
506
PART IGES DATA FILE
MANUALLY UNFOLD 3D PART INTO 2D VERSION
508
RECONSTRUCT 2D VERSION OUTLINE IN ELECTRICAL PACKAGE
510
LAYOUT ELECTRICAL CIRCUITRY
512
1

FIG. 5A

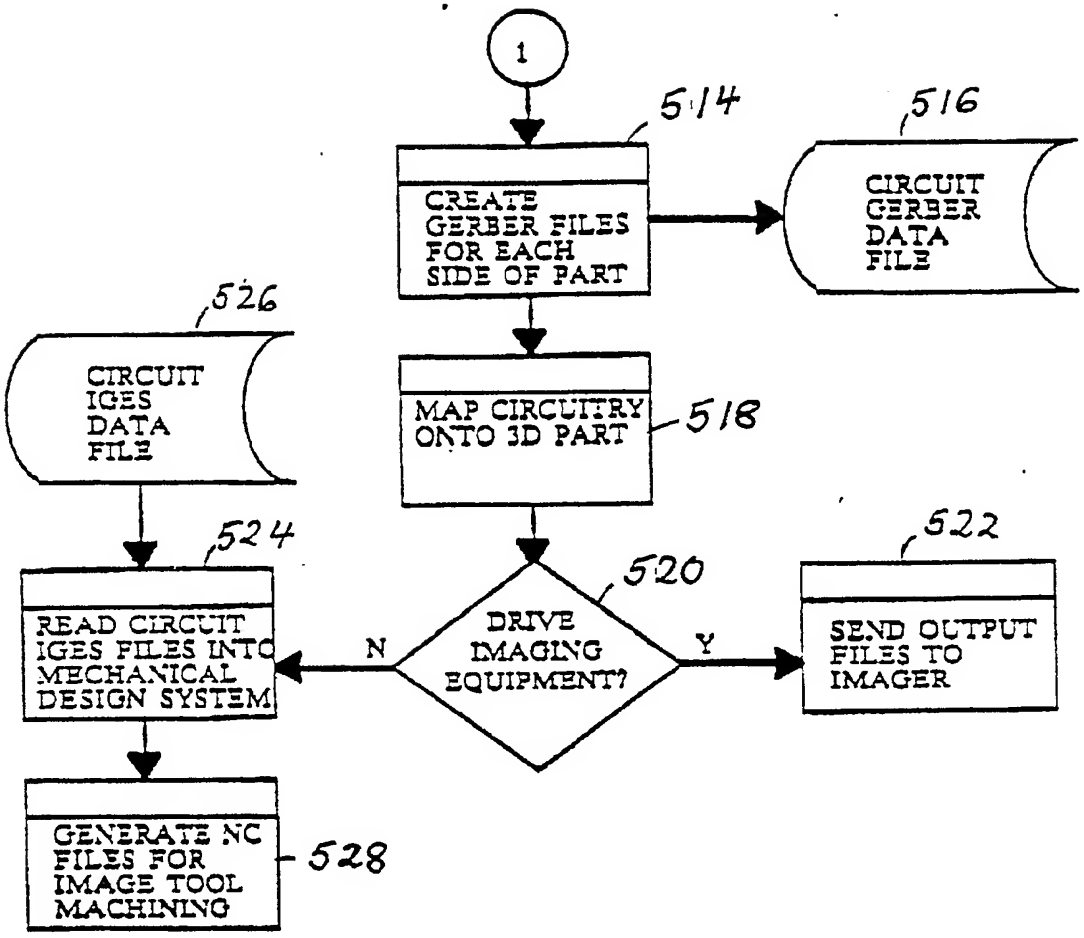
1
514
CREATE GERBER FILES FOR EACH SIDE OF PART
516
CIRCUIT GERBER DATA FILE
526
CIRCUIT IGES DATA FILE
MAP CIRCUITRY ONTO 3D PART
518
524
520
522
READ CIRCUIT IGES FILES INTO MECHANICAL DESIGN SYSTEM
N
DRIVE IMAGING EQUIPMENT?
Y
SEND OUTPUT FILES TO IMAGER
GENERATE NC FILES FOR IMAGE TOOL MACHINING
528

FIG. 5B

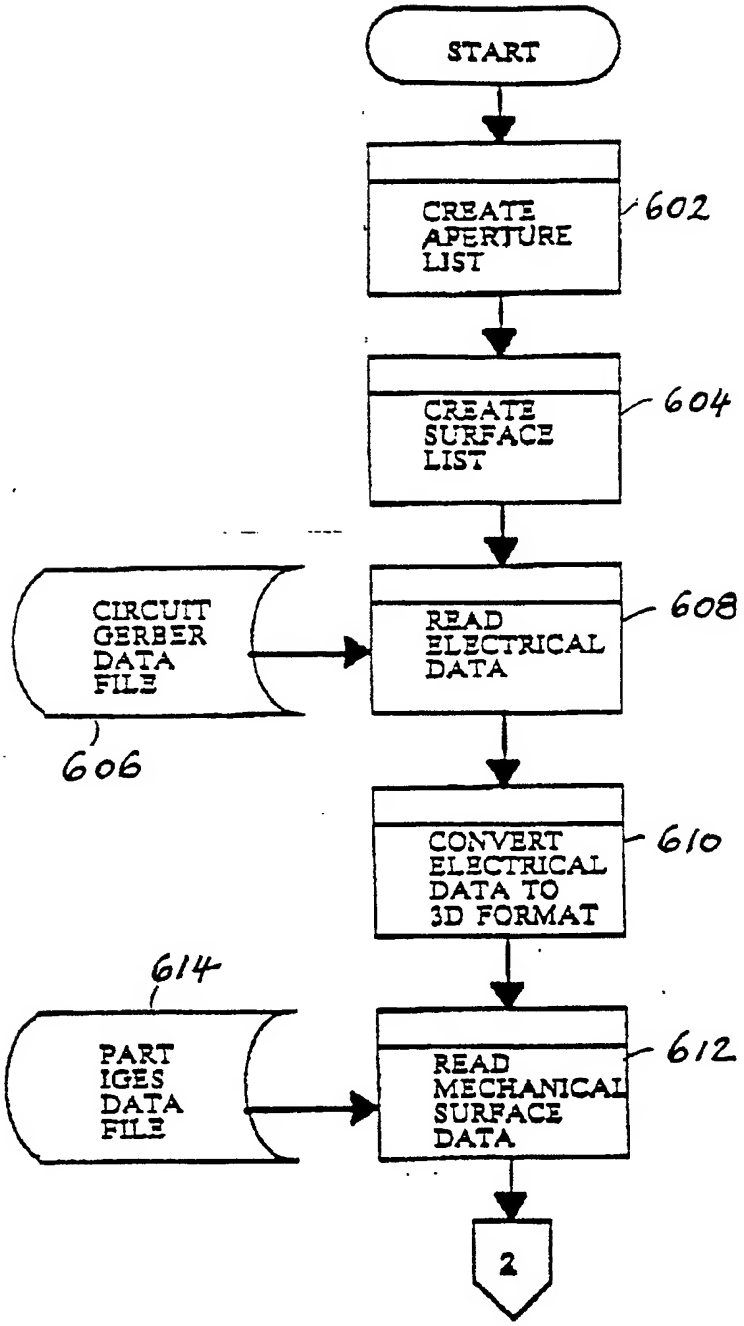
START
CREATE APERTURE LIST
602
CREATE SURFACE LIST
604
CIRCUIT GERBER DATA FILE
606
READ ELECTRICAL DATA
608
CONVERT ELECTRICAL DATA TO 3D FORMAT
610
614
PART IGES DATA FILE
READ MECHANICAL SURFACE DATA
612
2

FIG. 6A

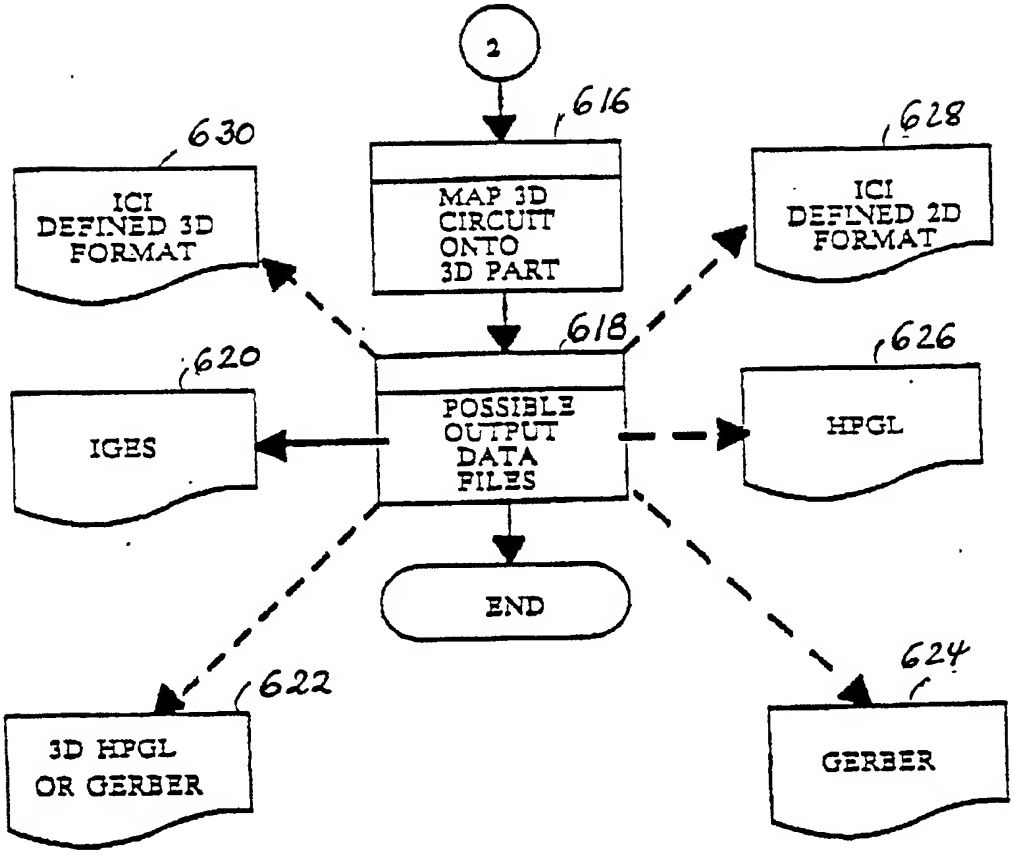
2
616
MAP 3D CIRCUIT ONTO 3D PART
618
POSSIBLE OUTPUT DATA FILES
END
630
ICI DEFINED 3D FORMAT
628
ICI DEFINED 2D FORMAT
620
IGES
626
HPGL
622
3D HPGL OR GERBER
624
GERBER

FIG. 6B

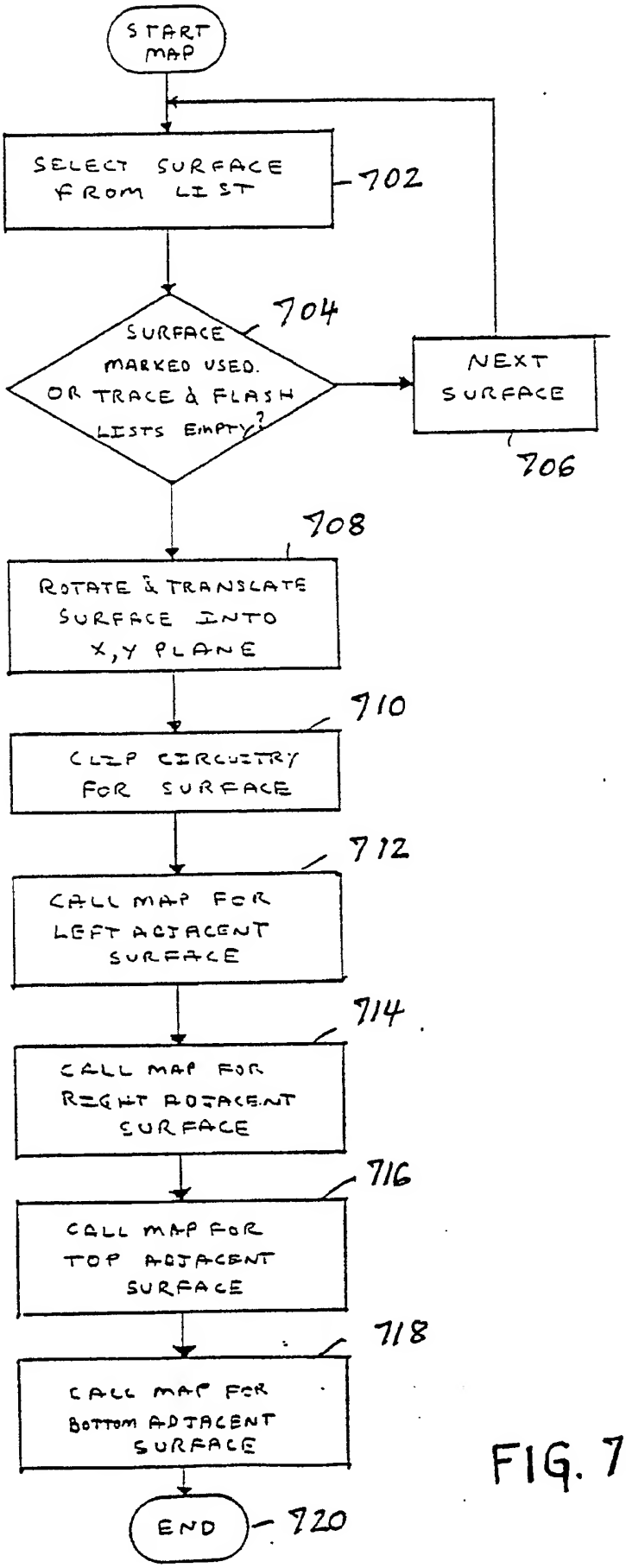
START MAP
SELECT SURFACE FROM LIST
702
SURFACE MARKED USED. OR TRACE & FLASH LISTS EMPTY?
704
NEXT SURFACE
706
ROTATE & TRANSLATE SURFACE INTO X,Y PLANE
708
CLIP CIRCUITRY FOR SURFACE
710
CALL MAP FOR LEFT ADJACENT SURFACE
712
CALL MAP FOR RIGHT ADJACENT SURFACE
714
CALL MAP FOR TOP ADJACENT SURFACE
716
CALL MAP FOR BOTTOM ADJACENT SURFACE
718
END
720

FIG. 7

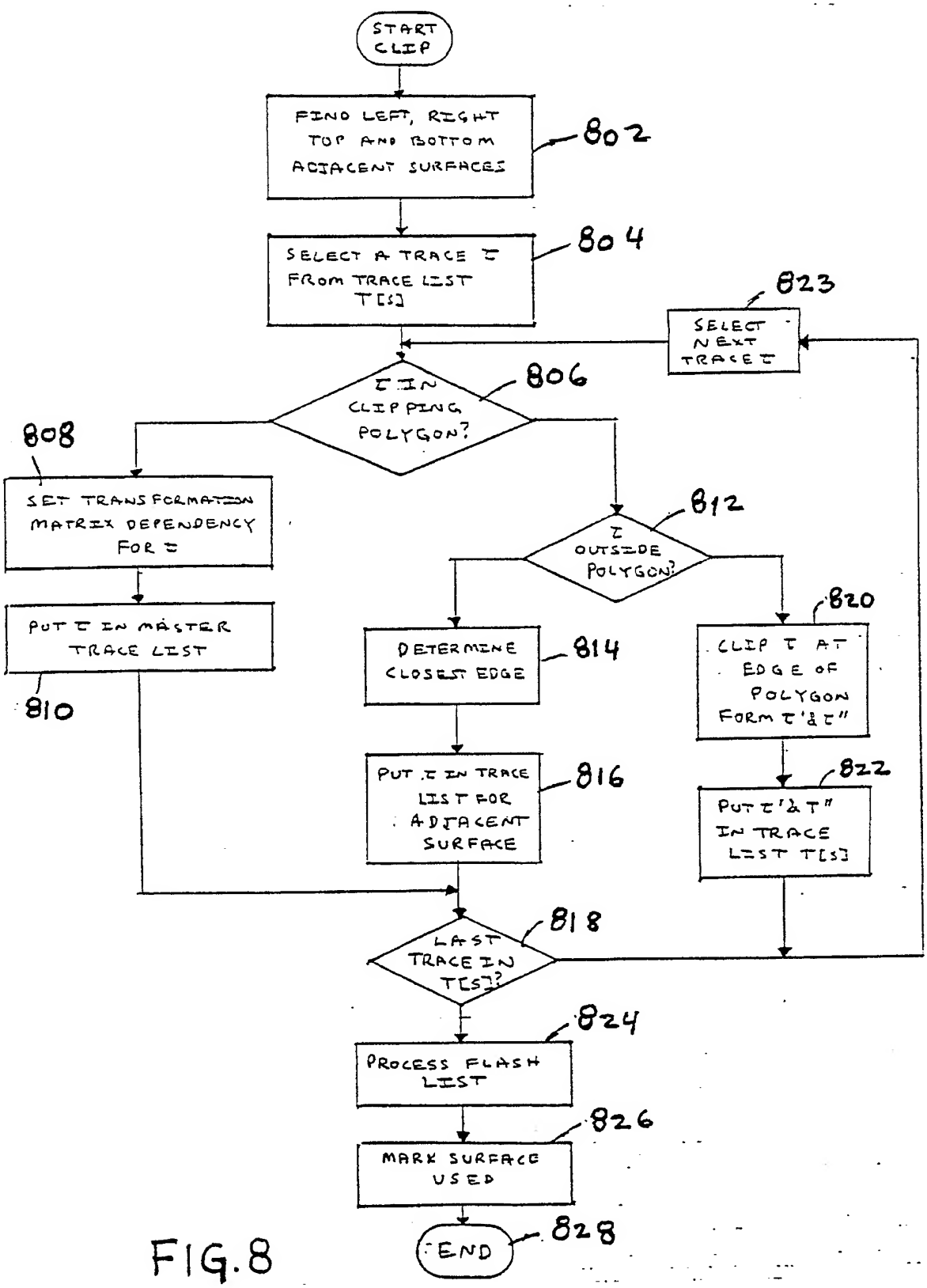
START CLIP
FIND LEFT, RIGHT TOP AND BOTTOM ADJACENT SURFACES
802
SELECT A TRACE T FROM TRACE LIST T[s]
804
SELECT NEXT TRACE T
823
T IN CLIPPING POLYGON?
806
808
SET TRANSFORMATION MATRIX DEPENDENCY FOR T
PUT T IN MASTER TRACE LIST
810
T OUTSIDE POLYGON?
812
DETERMINE CLOSEST EDGE
814
CLIP T AT EDGE OF POLYGON FORM T' & T"
820
PUT T IN TRACE LIST FOR ADJACENT SURFACE
816
PUT T' & T" IN TRACE LIST T[s]
822
LAST TRACE IN T[s]?
818
PROCESS FLASH LIST
824
MARK SURFACE USED
826
END
828

FIG.8